Lending

Smart Contract Audit Report
Prepared for FWX

Date Issued:	Jul 7, 2023
Project ID:	AUDIT2022036
Version:	v1.2
Confidentiality Level:	Public

CYBERSECURITY PROFESSIONAL SERVICE

Report Information

Project ID	AUDIT2022036
Version	v1.2
Client	FWX
Project	Lending
Auditor(s)	Patipon Suwanbol Puttimet Thammasaeng Wachirawit Kanpanluk
Author(s)	Patipon Suwanbol Wachirawit Kanpanluk Kongkit Chatchawanhirun
Reviewer	Natsasit Jirathammanuwat
Confidentiality Level	Public

Version History

Version	Date	Description	Author(s)
1.2	Jul 7, 2023	Update the token name from forward to FWX	Kongkit Chatchawanhirun
1.1	May 11, 2023	Update scope	Wachirawit Kanpanluk
1.0	Feb 8, 2023	Full report	Patipon Suwanbol Wachirawit Kanpanluk

Contact Information

Company	Inspex
Phone	(+66) 90 888 7186
Telegram	t.me/inspexco
Email	audit@inspex.co

Table of Contents

1. Executive Summary 1
1.1. Audit Result 1
1.2. Disclaimer 1
2. Project Overview 2
2.1. Project Introduction 2
2.2. Scope 3
3. Methodology 5
3.1. Test Categories 5
3.2. Audit Items 6
3.3. Risk Rating 8
4. Summary of Findings 9
5. Detailed Findings Information 12
5.1. Design Flaw in repay() Function 12
5.2. Denial of Service on Native Token Transfer to APHPool Contract 16
5.3. Price Manipulation in _rollover() Function 19
5.4. Centralized Control of State Variable 24
5.5. Use of Upgradable Contract Design 27
5.6. Design Flaw in Kill Switch Mechanism 30
5.7. Denial of Service in PoolLending 32
5.8. Denial of Service in Allowance Mechanism of InterestVault 35
5.9. Position Liquidation Avoidance 40
5.10. Design Flaw in Liquidation Mechanism 44
5.11. Transaction Ordering Dependence in _liquidationSwap() Function 50
5.12. Unnecessary Rate Validation in _queryRateUSD() Function 54
5.13. Incorrect State Variable Setting in setMembershipAddress() Function 57
5.14. Improper Interest Reward Distribution 59
5.15. Smart Contract with Unpublished Source Code 62
5.16. Missing Input Validation in setRankInfo() Function 63
5.17. Insufficient Logging for Privileged Functions 65
5.18. Outdated Compiler Version 67
5.19. Incorrect Logging for _withdraw() Function 70
5.20. Unnecessary Functions in InterestVault 73
5.21. Incorrect Logging for setForwAddress() Function 74
5.22. Improper Function Visibility 76

6. Appendix **78**
6.1. About Inspex 78

1. Executive Summary

As requested by FWX, Inspex team conducted an audit to verify the security posture of the Lending smart contracts between Jun 23, 2022 and Jul 5, 2022. During the audit, Inspex team examined all smart contracts and the overall operation within the scope to understand the overview of Lending smart contracts. Static code analysis, dynamic analysis, and manual review were done in conjunction to identify smart contract vulnerabilities together with technical & business logic flaws that may be exposed to the potential risk of the platform and the ecosystem. Practical recommendations are provided according to each vulnerability found and should be followed to remediate the issue.

1.1. Audit Result

In the initial audit, Inspex found 2 critical, 4 high, 5 medium, 4 low, 3 very low, and 4 info-severity issues. With the project team's prompt response, 2 critical, 2 high, 5 medium, 2 low, 2 very low, and 2 info-severity issues were resolved or mitigated in the reassessment, while 2 high, 2 low, 1 very low and 2 info-severity issues were acknowledged by the team. However, in the long run, Inspex suggests resolving all issues found in this report.

1.2. Disclaimer

This security audit is not produced to supplant any other type of assessment and does not guarantee the discovery of all security vulnerabilities within the scope of the assessment. However, we warrant that this audit is conducted with goodwill, professional approach, and competence. Since an assessment from one single party cannot be confirmed to cover all possible issues within the smart contract(s), Inspex suggests conducting multiple independent assessments to minimize the risks. Lastly, nothing contained in this audit report should be considered as investment advice.

2. Project Overview

2.1. Project Introduction

FWX is a decentralized finance platform on the EVM-based chain offering two main services: a decentralized derivative exchange (DDEX) and lending & borrowing pools (LBPs).

Defi-protocol-LBPs have served the asset's actual borrowing demand from users. The users can lend and borrow tokens from Defi-protocol-LBPs. The APRs are algorithmically determined by the demand and supply of the tokens. The higher the demand for borrowing, the higher the APRs will be. The lending interest is auto-compounded. In addition to the base lending APRs, some fees are shared among liquidity providers.

Scope Information:

Project Name	Lending
Website	https://fwx.finance/
Smart Contract Type	Ethereum Smart Contract
Chain	BNB Smart Chain
Programming Language	Solidity
Category	Lending

Audit Information:

Audit Method	Whitebox
Audit Date	Jun 23, 2022 - Jul 5, 2022
Reassessment Date	Nov 24, 2022

The audit method can be categorized into two types depending on the assessment targets provided:

1. **Whitebox**: The complete source code of the smart contracts are provided for the assessment.
2. **Blackbox**: Only the bytecodes of the smart contracts are provided for the assessment.

2.2. Scope

The following smart contracts were audited and reassessed by Inspex in detail:

Initial Audit

Contract	Bytecode SHA256 Hash
APHCore	88bd9b4fe29c6b3dd6b0db88b4bcaa7eb42ba2f80b3b311a89045bd14b7ddd99
APHCoreProxy	c43ca4004122d21bf9b78e15bc6e48092c97378bbbb483c8d33d90404ae26c18
CoreBase	2837cafb1a40c2a6463c657c9fe5fae2bc183b596cda33a1bcb00cad44e790e3
CoreBaseFunc	6e9b66984d912851e3e7e8769c678d0134157bcbc021d88872ea95efa1bc232c
CoreBorrowing	7b5a230a09382547f896731eafc4dd2cdb4169f20ca7168b1601f76586af5579
CoreSetting	919dc1373ab49a6e7eaa51c65b9e7328f1dae3c6746e1e8ef13514ff02272847
Timelock	23790898b67a316a8343e7324c88e9aa20c5bb633bbd6de6e9f2efb861557aad
Membership	117e48aaf98b30d17cdb88c3aab5d21c63468e1ac2abd6555b81477fc75e1d09
APHPool	12a414dd70a4e2125169eed873c56bfcee290b75b9a364f8903c05ca5e40b98e
APHPoolProxy	9bdfcaae841985c7bc87accd918c9d72f2baa34eedebd4d0db23e1481210ccb6
InterestVault	51933c32c15a95da7982320e26a16ed4f25db868ce4f126a7e0610112759e88c
PoolBorrowing	4e0b856f409d6da114b23270f30b1bb54e5fc060ee1322ab1aba0a6d7e9f25e2
PoolBaseFunc	a0c0dd7598764acab2b151da6d56a2e15793fd65e34f8c589b530557e45c861d
PoolLending	95a84ca5d116ae220623f18cd66402e025744c690a7a2876b2334a4ae3aebc1d
PoolToken	2dd4587258e5661cea352822acf3db3956cf90a05dc3fc0587f0ada03ed1ef73
PoolSetting	22df84a528809cf1ae06d9a0e01e2f3df798dde8d1ff75dd7f92db7c9b94eacb
StakePool	0fa9b82ce365308a9a6c53d95bf3abf21d91137d8fd2dbd4889982430838cc2d
PriceFeeds_BSC	e93e796712206dbba74a3ec17ec097d7975acf02dea0db75f900535c5422a1d6
ProxyAdmin	0ad40e63367024d812aa1e8628202c8a75d36a4b9735622f8a85b7d2cd6809f6
TransperantProxy	df9e798fa725c36c39da228f71f40f1986fd937137c18c2bf24fe85a79a33e71
Vault	a4ccf9eb23389d50b790a68cfbe89bc40a0d083ac264c1da9fedc8ea72bafae6
WETHHandler	6d5fb5c0b90c6dcf8869a2ec1f1ceb5cdb36c651a8f44adecc1f35712e83d505

Reassessment

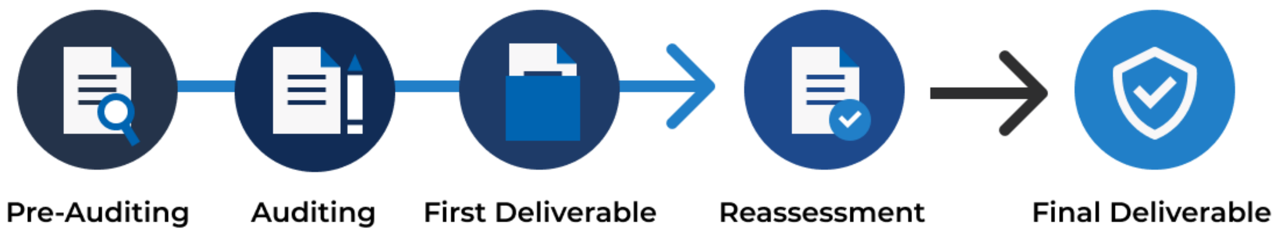
Contract	Bytecode SHA256 Hash
APHCore	f5cb0adf3249b7753b9d4fcd3131bc9bb6e906b94141bd903b16f88ac2c62dc7
APHCoreProxy	0d48fbb3ee5a65ef245300279737b4f3d3e8c31c7fc359ab56115c95c630627d
CoreBase	2661eb8be71a4f0822e2911a45f6a65516b3a2b879f1fd3d3923e6a292ecb358
CoreBaseFunc	ff388886d5aebd0a25cc0aea02b6b9a86e26d2b658267881ccc42a40e9e518d8
CoreBorrowing	9345a12951e616cfc51360d5cb5ea47d76f7fd90563a62f36aa679f3aad6a38e
CoreSetting	b77cccc06e91204bb33c57146298186f132d284fdafd9ec68753b7cf7802cb76
Timelock	66b862e48f05571c4f7883e25b6bd5a861e8e29777989f7f2f73bef45d30116f
Membership	7d303f71c008d11a9b95aa68b6a27cf100620d62476dfd89f3153c6bed6232bc
APHPool	d5484c572ab8b69f5947f71d3f6148a58cb9dcc95db4bc1a1e198be0cb7ef880
APHPoolProxy	d11cdbcbefdc4481d12b552001bc1c1f7e7192d83c48cd9d409337450f8f3952
InterestVault	4042ec1a698a0077e0ddd85e1419573553696389b48e3672bdd7cb1d89ad1bda
PoolBorrowing	7d129e081b7638991bfc8a0480131f9460906222e2e6cbb20e16fd4f5b3d50a3
PoolLending	61a8c2f7ea0da2cb0a46e955e4234c0dbb1181d418eedf9c090facd0f32094c3
PoolToken	5b45711027b1b1bcf0922e042cfac6f8786fab11ac451eb71709054119a4ffc3
PoolSetting	9e84ab11505d880ee8a3f034a49c3e9f695d4415a001a44c033d416419c0c84a
StakePool	f1fbad80ccb30d9eebe659d4e474af7917e0c6dac4b124a354caeeaf6211803f
PriceFeeds_BSC	293cd0946a710d277faa0a2f3a0b1cdd05a358433e6982bde73a83d7579a34ba
ProxyAdmin	61def3d777e296d7694cd28b025fff14617edf79fd2811f97f2bfb32e1116483
TransperantProxy	5647897a7cf9db8b6f72041349777a630239e76ae3698a2cb1c455ec3c6ba138
Vault	a41031a7f9aeef0fe8d838f4477f3f7138579eb31533bea132911db4242d9d96
WETHHandler	e4aee1d3a2da40142b0b24afa7eb2899d4872f9fd44aa9245a7b32f5bfd9f3d7

At a time of this report, the contracts have not been published yet. The user should compare the bytecode hashes with the smart contracts deployed before interacting with them to make sure that they are the same with the contracts audited

3. Methodology

Inspex conducts the following procedure to enhance the security level of our clients' smart contracts:

1. **Pre-Auditing**: Getting to understand the overall operations of the related smart contracts, checking for readiness, and preparing for the auditing
2. **Auditing**: Inspecting the smart contracts using automated analysis tools and manual analysis by a team of professionals
3. **First Deliverable and Consulting**: Delivering a preliminary report on the findings with suggestions on how to remediate those issues and providing consultation
4. **Reassessment**: Verifying the status of the issues and whether there are any other complications in the fixes applied
5. **Final Deliverable**: Providing a full report with the detailed status of each issue



3.1. Test Categories

Inspex smart contract auditing methodology consists of both automated testing with scanning tools and manual testing by experienced testers. We have categorized the tests into 3 categories as follows:

1. **General Smart Contract Vulnerability (General)** - Smart contracts are analyzed automatically using static code analysis tools for general smart contract coding bugs, which are then verified manually to remove all false positives generated.
2. **Advanced Smart Contract Vulnerability (Advanced)** - The workflow, logic, and the actual behavior of the smart contracts are manually analyzed in-depth to determine any flaws that can cause technical or business damage to the smart contracts or the users of the smart contracts.
3. **Smart Contract Best Practice (Best Practice)** - The code of smart contracts is then analyzed from the development perspective, providing suggestions to improve the overall code quality using standardized best practices.

3.2. Audit Items

The testing items checked are based on our Smart Contract Security Testing Guide (SCSTG) v1.0 (https://github.com/InspexCo/SCSTG/releases/download/v1.0/SCSTG_v1.0.pdf) which covers most prevalent risks in smart contracts. The latest version of the document can also be found at https://inspex.gitbook.io/testing-guide/.

The following audit items were checked during the auditing activity:

Testing Category	Testing Items
1. Architecture and Design	1.1. Proper measures should be used to control the modifications of smart contract logic 1.2. The latest stable compiler version should be used 1.3. The circuit breaker mechanism should not prevent users from withdrawing their funds 1.4. The smart contract source code should be publicly available 1.5. State variables should not be unfairly controlled by privileged accounts 1.6. Least privilege principle should be used for the rights of each role
2. Access Control	2.1. Contract self-destruct should not be done by unauthorized actors 2.2. Contract ownership should not be modifiable by unauthorized actors 2.3. Access control should be defined and enforced for each actor roles 2.4. Authentication measures must be able to correctly identify the user 2.5. Smart contract initialization should be done only once by an authorized party 2.6. tx.origin should not be used for authorization
3. Error Handling and Logging	3.1. Function return values should be checked to handle different results 3.2. Privileged functions or modifications of critical states should be logged 3.3. Modifier should not skip function execution without reverting
4. Business Logic	4.1. The business logic implementation should correspond to the business design 4.2. Measures should be implemented to prevent undesired effects from the ordering of transactions 4.3. msg.value should not be used in loop iteration
5. Blockchain Data	5.1. Result from random value generation should not be predictable 5.2. Spot price should not be used as a data source for price oracles 5.3. Timestamp should not be used to execute critical functions 5.4. Plain sensitive data should not be stored on-chain 5.5. Modification of array state should not be done by value 5.6. State variable should not be used without being initialized

Testing Category	Testing Items
6. External Components	6.1. Unknown external components should not be invoked 6.2. Funds should not be approved or transferred to unknown accounts 6.3. Reentrant calling should not negatively affect the contract states 6.4. Vulnerable or outdated components should not be used in the smart contract 6.5. Deprecated components that have no longer been supported should not be used in the smart contract 6.6. Delegatecall should not be used on untrusted contracts
7. Arithmetic	7.1. Values should be checked before performing arithmetic operations to prevent overflows and underflows 7.2. Explicit conversion of types should be checked to prevent unexpected results 7.3. Integer division should not be done before multiplication to prevent loss of precision
8. Denial of Services	8.1. State changing functions that loop over unbounded data structures should not be used 8.2. Unexpected revert should not make the whole smart contract unusable 8.3. Strict equalities should not cause the function to be unusable
9. Best Practices	9.1. State and function visibility should be explicitly labeled 9.2. Token implementation should comply with the standard specification 9.3. Floating pragma version should not be used 9.4. Builtin symbols should not be shadowed 9.5. Functions that are never called internally should not have public visibility 9.6. Assert statement should not be used for validating common conditions

3.3. Risk Rating

OWASP Risk Rating Methodology (https://owasp.org/www-community/OWASP_Risk_Rating_Methodology) is used to determine the severity of each issue with the following criteria:

- **Likelihood**: a measure of how likely this vulnerability is to be uncovered and exploited by an attacker
- **Impact**: a measure of the damage caused by a successful attack

Both likelihood and impact can be categorized into three levels: **Low**, **Medium**, and **High**.

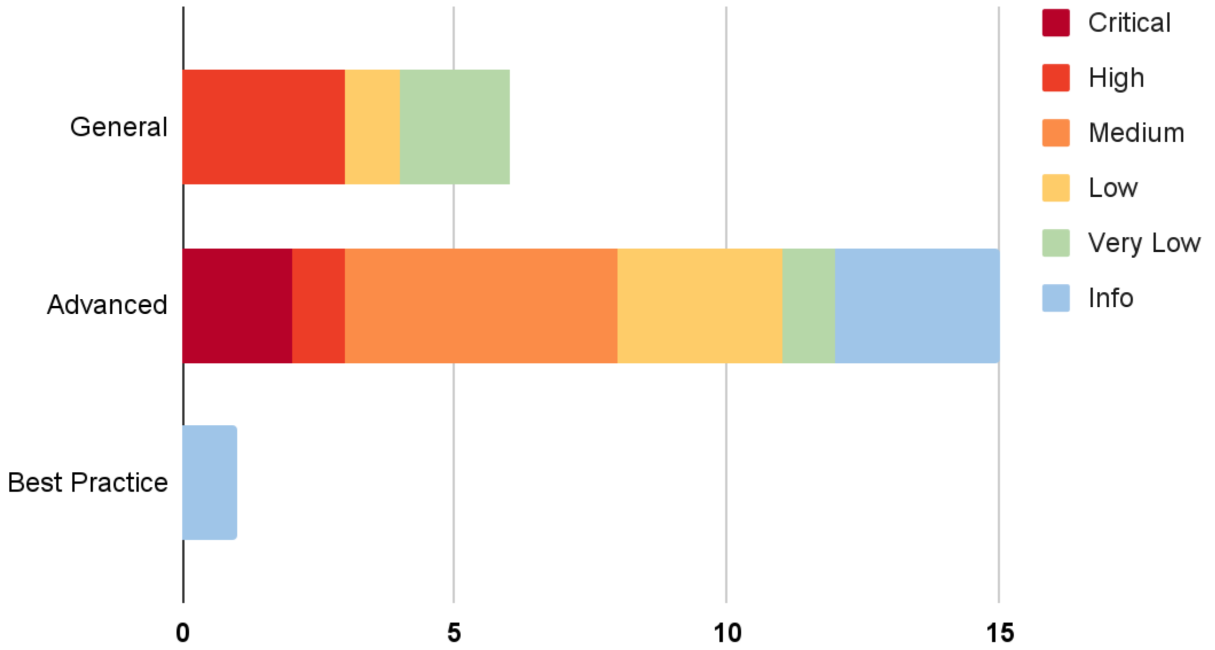
Severity is the overall risk of the issue. It can be categorized into five levels: **Very Low**, **Low**, **Medium**, **High**, and **Critical**. It is calculated from the combination of likelihood and impact factors using the matrix below. The severity of findings with no likelihood or impact would be categorized as **Info**.

Likelihood Impact	Low	Medium	High
Low	Very Low	Low	Medium
Medium	Low	Medium	High
High	Medium	High	Critical

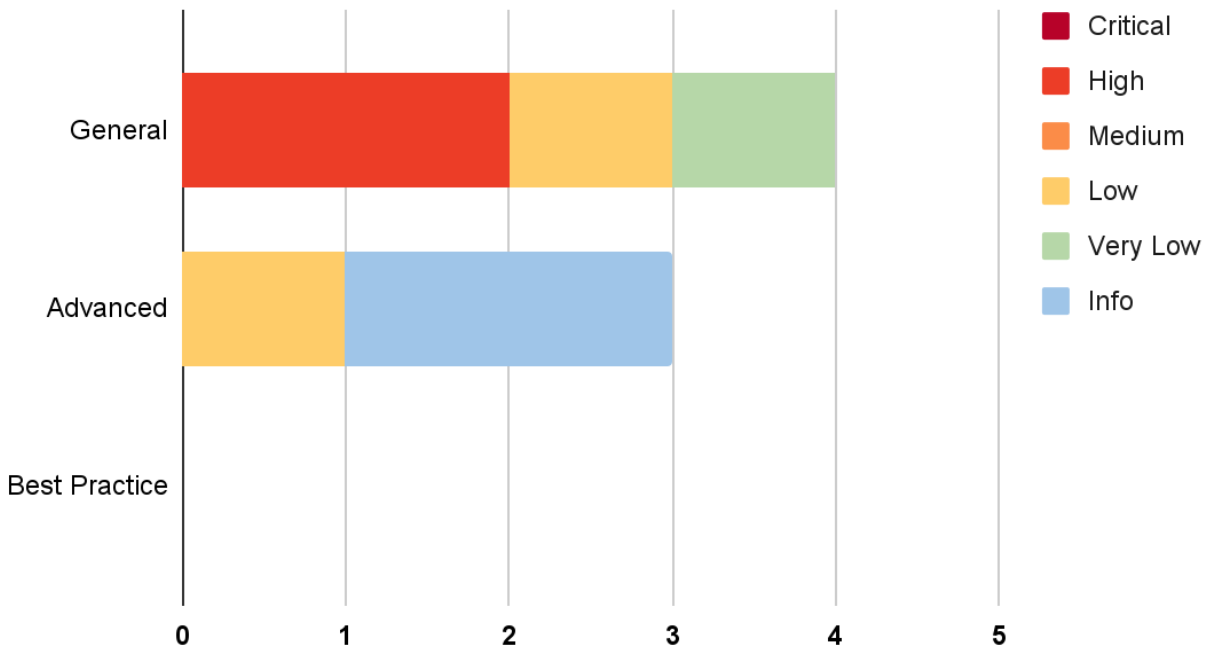
4. Summary of Findings

The following charts show the number of the issues found during the assessment and the issues acknowledged in the reassessment, categorized into three categories: **General**, **Advanced**, and **Best Practice**.

Assessment:



Reassessment:



The statuses of the issues are defined as follows:

Status	Description
Resolved	The issue has been resolved and has no further complications.
Resolved *	The issue has been resolved with mitigations and clarifications. For the clarification or mitigation detail, please refer to Chapter 5.
Acknowledged	The issue's risk has been acknowledged and accepted.
No Security Impact	The best practice recommendation has been acknowledged.

The information and status of each issue can be found in the following table:

ID	Title	Category	Severity	Status
IDX-001	Design Flaw in repay() Function	Advanced	Critical	Resolved
IDX-002	Denial of Service on Native Token Transfer to APHPool Contract	Advanced	Critical	Resolved
IDX-003	Price Manipulation in _rollover() Function	Advanced	High	Resolved
IDX-004	Centralized Control of State Variable	General	High	Acknowledged
IDX-005	Use of Upgradable Contract Design	General	High	Resolved *
IDX-006	Design Flaw in Kill Switch Mechanism	General	High	Acknowledged
IDX-007	Denial of Service in PoolLending	Advanced	Medium	Resolved *
IDX-008	Denial of Service in Allowance Mechanism of InterestVault	Advanced	Medium	Resolved
IDX-009	Position Liquidation Avoidance	Advanced	Medium	Resolved
IDX-010	Design Flaw in Liquidation Mechanism	Advanced	Medium	Resolved
IDX-011	Transaction Ordering Dependence in _liquidationSwap() Function	Advanced	Medium	Resolved *
IDX-012	Unnecessary Rate Validation in _queryRateUSD() Function	Advanced	Low	Resolved
IDX-013	Incorrect State Variable Setting in setMembershipAddress() Function	Advanced	Low	Resolved
IDX-014	Improper Interest Reward Distribution	Advanced	Low	Acknowledged

IDX-015	Smart Contract with Unpublished Source Code	General	Low	Acknowledged
IDX-016	Missing Input Validation in setRankInfo() Function	Advanced	Very Low	Resolved
IDX-017	Insufficient Logging for Privileged Functions	General	Very Low	Resolved
IDX-018	Outdated Compiler Version	General	Very Low	Acknowledged
IDX-019	Incorrect Logging for _withdraw() Function	Advanced	Info	No Security Impact
IDX-020	Unnecessary Functions in InterestVault	Advanced	Info	No Security Impact
IDX-021	Incorrect Logging for setForwAddress() Function	Advanced	Info	Resolved
IDX-022	Improper Function Visibility	Best Practice	Info	Resolved

* The mitigations or clarifications by FWX can be found in Chapter 5.

5. Detailed Findings Information

5.1. Design Flaw in repay() Function

ID	IDX-001
Target	CoreBorrowing
Category	Advanced Smart Contract Vulnerability
CWE	CWE-755: Improper Handling of Exceptional Conditions
Risk	**Severity: Critical** **Impact: High** The users will not be able to repay the loan position, causing monetary loss for the users, disruption of service, and loss of reputation to the platform. **Likelihood: High** This issue will occur every time when the users repay the borrowed token with $WETH.
Status	Resolved The FWX team has resolved this issue by using the `ERC20.safeTransfer()` function instead of the `_transferFromIn()` function after wrapping native tokens to transfer both `borrowPaid` and `interestPaid`.

5.1.1. Description

In the `CoreBorrowing` contract, the `repay()` function calls the `_transferFromIn()` function to repay the interest of the loan position.

When the borrowed token is wrapped native token, the token will be transferred from the user in 3 portions, `msg.value`, `borrowPaid`, and `interestPaid`, exceeding the total value of `repayAmount` as shown below in lines 81 - 85:

CoreBorrowing.sol

```
48 function repay(
49     uint256 loanId,
50     uint256 nftId,
51     uint256 repayAmount,
52     bool isOnlyInterest
53 )
54     external
55     payable
56     whenFuncNotPaused(msg.sig)
57     nonReentrant
58     returns (uint256 borrowPaid, uint256 interestPaid)
```

```
{
    nftId = _getUsableToken(msg.sender, nftId);
    Loan storage loan = loans[nftId][loanId];
    require(
        loan.borrowTokenAddress == wethAddress || msg.value == 0,
        "CoreBorrowing/no-support-transfering-ether-in"
    );

    bool isLoanClosed;
    uint256 tmpCollateralAmount = loan.collateralAmount;
    (borrowPaid, interestPaid, isLoanClosed) = _repay(
        loanId,
        nftId,
        repayAmount,
        isOnlyInterest
    );

    if (loan.borrowTokenAddress == wethAddress) {
        require(
            msg.value >= borrowPaid + interestPaid,
            "CoreBorrowing/insufficient-ether-amount"
        );
        _transferFromIn(msg.sender, address(this), wethAddress, msg.value);
        if (borrowPaid > 0) {
            _transferFromIn(msg.sender, assetToPool[wethAddress], wethAddress, borrowPaid);
        }
        _transferFromIn(
            msg.sender,
            IAPHPool(assetToPool[wethAddress]).interestVaultAddress(),
            wethAddress,
            interestPaid
        );
        _transferOut(msg.sender, wethAddress, msg.value - (borrowPaid + interestPaid));
    } else {
        if (borrowPaid > 0) {
            _transferFromIn(
                msg.sender,
                assetToPool[loan.borrowTokenAddress],
                loan.borrowTokenAddress,
                borrowPaid
            );
        }
        _transferFromIn(
            msg.sender,
```

```
    IAPHPool(assetToPool[loan.borrowTokenAddress]).interestVaultAddress(),
104                 loan.borrowTokenAddress,
105                 interestPaid
106             );
107         }
108         if (isLoanClosed) {
109             _transferOut(msg.sender, loan.collateralTokenAddress,
    tmpCollateralAmount);
110         }
111         return (borrowPaid, interestPaid);
112     }
```

Furthermore, from the source code above in line 77, the `msg.value` has to be greater than or equal to the sum of `borrowPaid` and `interestPaid` to call the `_transferFromIn()` function in lines 83 and 85. The `amount` parameter, passed in from the `interestPaid` and `borrowPaid` values, has to be equal to the `msg.value`, which conflicts with the condition in the `repay()` function, causing this function to be reverted as shown below in line 36:

AssetHandlerUpgradeable.sol

```
27  function _transferFromIn(
28      address from,
29      address to,
30      address token,
31      uint256 amount
32  ) internal {
33      require(amount != 0, "AssetHandler/amount-is-zero");
34  
35      if (token == wethAddress) {
36          require(amount == msg.value, "AssetHandler/value-not-matched");
37          IWethERC20(wethAddress).deposit{value: amount}();
38          IWethERC20(wethAddress).safeTransfer(to, amount);
39      } else {
40          IERC20(token).safeTransferFrom(from, to, amount);
41      }
42  }
```

Without being able to repay for the loan position, the users cannot get their collateral tokens back, losing that portion of funds to the contract.

5.1.2. Remediation

Inspex suggests modifying the logic of the `repay()` function to be using wrapped native token, and replace all native token transfers with ERC20 transfer instead, for example:

CoreBorrowing.sol

```
function repay(
    uint256 loanId,
    uint256 nftId,
    uint256 repayAmount,
    bool isOnlyInterest
)
    external
    whenFuncNotPaused(msg.sig)
    nonReentrant
    returns (uint256 borrowPaid, uint256 interestPaid)
{
    nftId = _getUsableToken(msg.sender, nftId);
    Loan storage loan = loans[nftId][loanId];

    bool isLoanClosed;
    uint256 tmpCollateralAmount = loan.collateralAmount;
    (borrowPaid, interestPaid, isLoanClosed) = _repay(
        loanId,
        nftId,
        repayAmount,
        isOnlyInterest
    );

    if (borrowPaid > 0) {
        IERC20(loan.borrowTokenAddress).safeTransferFrom(msg.sender,
assetToPool[loan.borrowTokenAddress], borrowPaid);
    }
    IERC20(loan.borrowTokenAddress).safeTransferFrom(msg.sender,
IAPHPool(assetToPool[loan.borrowTokenAddress]).interestVaultAddress(),
interestPaid);

    if (isLoanClosed) {
        IERC20(loan.collateralTokenAddress).safeTransfer(msg.sender,
tmpCollateralAmount);
    }
    return (borrowPaid, interestPaid);
}
```

5.2. Denial of Service on Native Token Transfer to APHPool Contract

ID	IDX-002
Target	APHPool
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity: Critical** **Impact: High** The position with wrapped native token as a borrowed asset will not be able to be liquidated. **Likelihood: High** It is very likely that the position with borrowed assets as a wrapped native token will not be able to be liquidated as the target contract address does not have `receive()` function to receive the native token.
Status	**Resolved** The FWX team has resolved this issue by transferring wrapped native token instead of native token.

5.2.1. Description

When a position is liquidated in the `CoreBorrowing` contract, the contract will swap the collateral asset to the borrowed asset and transfer the principal (`repayBorrow`) back to the lending pool as shown in line 192:

CoreBorrowing.sol

```
178 function liquidate(uint256 loanId, uint256 nftId)
179     external
180     whenFuncNotPaused(msg.sig)
181     nonReentrant
182     returns (
183         uint256 repayBorrow,
184         uint256 repayInterest,
185         uint256 bountyReward,
186         uint256 leftOverCollateral
187     )
188 {
189     Loan storage loan = loans[nftId][loanId];
190     (repayBorrow, repayInterest, bountyReward, leftOverCollateral) =
    _liquidate(loanId, nftId);
191
192     _transferOut(assetToPool[loan.borrowTokenAddress], loan.borrowTokenAddress,
    repayBorrow);
```

```
    _transferOut(
        IAPHPool(assetToPool[loan.borrowTokenAddress]).interestVaultAddress(),
        loan.borrowTokenAddress,
        repayInterest
    );

    _transferOut(msg.sender, loan.collateralTokenAddress, bountyReward);
    _transferOut(_getTokenOwnership(nftId), loan.collateralTokenAddress,
leftOverCollateral);
}
```

If the borrowed token is the wrapped native token (`loan.borrowTokenAddress`), the `_transferOut` function will unwrap it and transfer the native token to the target address as shown in line 69 to 71, in this case, it is the pool contract (`APHPool`).

AssetHandlerUpgradeable.sol

```
61 function _transferOut(
62     address to,
63     address token,
64     uint256 amount
65 ) internal {
66     if (amount == 0) {
67         return;
68     }
69     if (token == wethAddress) {
70         IWethERC20(wethAddress).safeTransfer(wethHandler, amount);
71         IWethHandler(payable(wethHandler)).withdrawETH(to, amount);
72     } else {
73         IERC20(token).safeTransfer(to, amount);
74     }
75 }
```

However, in the `APHPool` contract implementation, there is no `receive()` function nor `fallback()` function with `payable`. Therefore, the pool contract (`APHPool`) will not be able to receive the native token.

5.2.2. Remediation

Inspex suggests adding a `receive()` function to the implementation, a `APHPool` contract, to make it able to receive the native token.

APHPool.sol

```
receive() external payable {}
```

However, as consulted with the platform, they decided to use ERC20 token (including wrapped native token) for all the token transfers between all contracts, so the `_transferOut` function should be changed to the `IERC20(tokenAddress).safeTransfer()`. As a result, this issue will be resolved without the need to implement the `receive()` function.

5.3. Price Manipulation in _rollover() Function

ID	IDX-003
Target	CoreBorrowing
Category	Advanced Smart Contract Vulnerability
CWE	CWE-807: Reliance on Untrusted Inputs in a Security Decision
Risk	**Severity:** High **Impact:** High The asset price on the platform can be arbitrarily manipulated by the flashloan attack. An attacker can inflate the price and drain the collateral from any overdue position. **Likelihood:** Medium This attack can be very profitable for the attacker, so there is high incentive for the attack; however, this can only be done to the positions that are overdue.
Status	Resolved The FWX team has resolved this issue by applying price oracle when retrieving the borrow and collateral asset prices in the `_rollover()` function.

5.3.1. Description

In the `CoreBorrowing` contract, the `liquidate()` function calls `_liquidate()` function in line 190.

CoreBorrowing.sol

```
178 function liquidate(uint256 loanId, uint256 nftId)
179     external
180     whenFuncNotPaused(msg.sig)
181     nonReentrant
182     returns (
183         uint256 repayBorrow,
184         uint256 repayInterest,
185         uint256 bountyReward,
186         uint256 leftOverCollateral
187     )
188 {
189     Loan storage loan = loans[nftId][loanId];
190     (repayBorrow, repayInterest, bountyReward, leftOverCollateral) =
    _liquidate(loanId, nftId);
191
192     _transferOut(assetToPool[loan.borrowTokenAddress], loan.borrowTokenAddress,
    repayBorrow);
193     _transferOut(
194         IAPHPool(assetToPool[loan.borrowTokenAddress]).interestVaultAddress(),
```

```
        loan.borrowTokenAddress,
        repayInterest
    );

    _transferOut(msg.sender, loan.collateralTokenAddress, bountyReward);
    _transferOut(_getTokenOwnership(nftId), loan.collateralTokenAddress, leftOverCollateral);
}
```

The `_liquidate()` function then calls the `_rollover()` function when the loan is overdue as shown in line 576.

CoreBorrowing.sol

```
550 function _liquidate(uint256 loanId, uint256 nftId)
551     internal
552     returns (
553         uint256 repayBorrow,
554         uint256 repayInterest,
555         uint256 bountyReward,
556         uint256 leftOverCollateral
557     )
558 {
559     Loan storage loan = loans[nftId][loanId];
560     LoanConfig storage loanConfig = loanConfigs[loan.borrowTokenAddress][
561         loan.collateralTokenAddress
562     ];
563     // rate, precision, maxSwappable (collaToken => borrowToken)
564     uint256[] memory numberArray = new uint256[](3);
565     require(loanExts[nftId][loanId].active == true, "CoreBorrowing/loan-is-closed");
566 
567     _settleBorrowInterest(loan);
568 
569     (numberArray[0], numberArray[1]) = _queryRate(
570         loan.collateralTokenAddress,
571         loan.borrowTokenAddress
572     );
573 
574     // rollover if loan is overdue
575     if (block.timestamp > loan.rolloverTimestamp) {
576         (, bountyReward) = _rollover(loanId, nftId, msg.sender);
577     }
578 
579     // liquidate
580     if (
581         _isLoanLTVExceedTargetLTV(
582             loan.borrowAmount,
```

```
                loan.collateralAmount,
                MathUpgradeable.max(loan.interestOwed, loan.minInterest),
                loanConfig.liquidationLTV,
                numberArray[0],
                numberArray[1]
            )
        ) {
            (uint256 collateralTokenAmountUsed, uint256 borrowTokenAmountSwap) = _liquidationSwap(
                loan
            );

            leftOverCollateral = loan.collateralAmount - collateralTokenAmountUsed;

            (repayBorrow, repayInterest, ) = _repay(loanId, nftId, borrowTokenAmountSwap, false);

            if (loanExts[nftId][loanId].active == true) {
                // TODO (future work): handle with ciritical condition, this part must add pool subsidisation for pool loss
                // Ciritical condition, protocol loss transfer int or sth else to pool
            } else {
                bountyReward += (leftOverCollateral * loanConfig.bountyFeeRate) / WEI_PERCENT_UNIT;
                leftOverCollateral -=
                    (leftOverCollateral * loanConfig.bountyFeeRate) /
                    WEI_PERCENT_UNIT;
            }

            emit Liquidate(
                msg.sender,
                nftId,
                loanId,
                msg.sender,
                numberArray[0],
                borrowTokenAmountSwap,
                bountyReward,
                loan.collateralTokenAddress,
                leftOverCollateral
            );
        }
}
```

In the `_rollover()` function, the `IRouter(routerAddress).getAmountsOut()` function is called to get the price for the collateral as shown below in line 514:

CoreBorrowing.sol

```
function _rollover(
    uint256 loanId,
    uint256 nftId,
    address caller
) internal returns (uint256 delayInterest, uint256 collateralBountyReward) {
    Loan storage loan = loans[nftId][loanId];
    require(loanExts[nftId][loanId].active == true, "CoreBorrowing/loan-is-closed");
    uint256 bountyReward;

    _settleBorrowInterest(loan);

    LoanConfig storage loanConfig = loanConfigs[loan.borrowTokenAddress][
        loan.collateralTokenAddress
    ];

    // This loan is overdue
    if (block.timestamp > loan.rolloverTimestamp) {
        delayInterest = ((block.timestamp - loan.rolloverTimestamp) * loan.owedPerDay) / 1 days;
        bountyReward = (delayInterest * loanConfig.bountyFeeRate) / WEI_PERCENT_UNIT;

        if (caller == _getTokenOwnership(nftId)) {
            // Caller is owner, collect delay interest to interestOwed
            loan.interestOwed += delayInterest + bountyReward;
        } else {
            // Caller is liquidator, bounty fee is sent to liquidator in form of collateral token equal to bountyFee
            address[] memory path_data = new address[](2);
            path_data[0] = loan.borrowTokenAddress;
            path_data[1] = loan.collateralTokenAddress;

            collateralBountyReward = IRouter(routerAddress).getAmountsOut(
                bountyReward,
                path_data
            )[1];
            bountyReward = 0;
            loan.interestOwed += delayInterest;
            loan.collateralAmount -= collateralBountyReward;
        }
    }
    address poolAddress = assetToPool[loan.borrowTokenAddress];
    PoolStat storage poolStat = poolStats[poolAddress];

    (uint256 interestRate, ) = IAPHPool(poolAddress).calculateInterest(0);
```

```
    uint256 interestOwedPerDay = (loan.borrowAmount * interestRate) / (WEI_PERCENT_UNIT * 365);

    loan.rolloverTimestamp = uint64(block.timestamp + loanDuration);
    poolStat.borrowInterestOwedPerDay =
        poolStat.borrowInterestOwedPerDay -
        loan.owedPerDay +
        interestOwedPerDay;

    loan.owedPerDay = interestOwedPerDay;
    loan.lastSettleTimestamp = uint64(block.timestamp);

    emit Rollover(
        msg.sender,
        nftId,
        loanId,
        caller,
        delayInterest + bountyReward,
        collateralBountyReward,
        loan.collateralTokenAddress,
        interestOwedPerDay
    );
}
```

The `IRouter(routerAddress).getAmountsOut()` function is used to retrieve the current price of asset liquidity, which can be manipulated easily by the flashloan attack as an example.

Therefore, if a position is overdue, an attacker can perform a flashloan attack to massively inflate the `collateralBountyReward`, allowing the attacker to drain the collateral token from the loan position that is overdue.

5.3.2. Remediation

Inspex suggests using the price data from a trustable price oracle provider instead of using the spot price.

If the price of the needed assets are not available from other trustable sources, Inspex suggests using a time-weight average price instead of directly quoting from the reserves.

More information on time-weight average price can be found in the following link:
https://docs.uniswap.org/contracts/v2/concepts/core-concepts/oracles

5.4. Centralized Control of State Variable

ID	IDX-004
Target	CoreBaseFunc CoreSetting StakePool Vault Membership PoolSetting InterestVault PriceFeeds_BSC
Category	General Smart Contract Vulnerability
CWE	CWE-284: Improper Access Control
Risk	**Severity:** High **Impact:** High The controlling authorities can change the critical state variables to gain additional profit. Thus, it is unfair to the other users. **Likelihood:** Medium There is nothing to restrict the changes from being done; however, this action can only be done by the contract owner.
Status	Acknowledged The FWX team has acknowledged this issue. They will be applying the timelock as the solution to delay the privilege control effect, which is set at 12 hours as a minimum delay. However, with only 12 hours of delay, this might not be able to suit all the users' time zones, so we suggest applying the timelock delay for at least 24 hours.

5.4.1. Description

Critical state variables can be updated at any time by the controlling authorities. Changes in these variables can cause impacts to the users, so the users should accept or be notified before these changes are effective.

However, there is currently no constraint to prevent the authorities from modifying these variables without notifying the users.

The controllable privileged state update functions are as follows:

File	Contract	Function	Modifier
CoreSetting.sol (L:12)	CoreSetting	setMembershipAddress()	onlyManager

CoreSetting.sol (L:19)	CoreSetting	setPriceFeedAddress()	onlyManager
CoreSetting.sol (L:26)	CoreSetting	setForwDistributorAddress()	onlyManager
CoreSetting.sol (L:37)	CoreSetting	setRouterAddress()	onlyManager
CoreSetting.sol (L:44)	CoreSetting	setWETHHandler()	onlyManager
CoreSetting.sol (L:51)	CoreSetting	setCoreBorrowingAddress()	onlyManager
CoreSetting.sol (L:58)	CoreSetting	setFeeController()	onlyManager
CoreSetting.sol (L:66)	CoreSetting	setLoanDuration()	onlyManager
CoreSetting.sol (L:73)	CoreSetting	setAdvancedInterestDuration()	onlyManager
CoreSetting.sol (L:80)	CoreSetting	setFeeSpread()	onlyManager
CoreSetting.sol (L:88)	CoreSetting	registerNewPool()	onlyManager
CoreSetting.sol (L:110)	CoreSetting	setForwDisPerBlock()	onlyManager
CoreSetting.sol (L:140)	CoreSetting	setupLoanConfig()	-
CoreSetting.sol (L:191)	CoreSetting	approveForRouter()	onlyManager
Membership.sol (L:34)	Membership	setNewPool()	onlyOwner
Membership.sol (L:43)	Membership	setBaseURI()	onlyOwner
StakePool.sol (L:37)	StakePool	setNextPool()	onlyOwner
StakePool.sol (L:44)	StakePool	setSettleInterval()	onlyOwner
StakePool.sol (L:51)	StakePool	setSettlePeriod()	onlyOwner
StakePool.sol (L:58)	StakePool	setPoolStartTimestamp()	onlyOwner
StakePool.sol (L:72)	StakePool	setRankInfo()	onlyOwner
Vault.sol (L:19)	Vault	ownerApprove()	onlyOwner
Vault.sol (L:27)	Vault	approveInterestVault()	onlyOwner
Membership.sol (L:34)	Membership	setNewPool()	onlyOwner
Membership.sol (L:43)	Membership	setBaseURI()	onlyOwner
PoolSetting.sol (L:10)	PoolSetting	setBorrowInterestParams()	onlyManager
PoolSetting.sol (L:35)	PoolSetting	setupLoanConfig()	onlyManager

PoolSetting.sol (L:68)	PoolSetting	setPoolLendingAddress()	onlyManager
PoolSetting.sol (L:75)	PoolSetting	setPoolBorrowingAddress()	onlyManager
PoolSetting.sol (L:82)	PoolSetting	setWETHHandler()	onlyManager
PoolSetting.sol (L:89)	PoolSetting	setMembershipAddress()	onlyManager
InterestVault.sol (L:51)	InterestVault	setForwAddress()	onlyManager
InterestVault.sol (L:68)	InterestVault	setTokenAddress()	onlyManager
InterestVault.sol (L:75)	InterestVault	setProtocolAddress()	onlyManager
PriceFeed.sol (L:101)	PriceFeeds_BSC	setPriceFeed()	onlyOwner
PriceFeed.sol (L:110)	PriceFeeds_BSC	setDecimals()	onlyOwner
PriceFeed.sol (L:120)	PriceFeeds_BSC	setGlobalPricingPaused()	onlyOwner

5.4.2. Remediation

In the ideal case, the critical state variables should not be modifiable to keep the integrity of the smart contract. However, if modifications are needed, Inspex suggests implementing a community-run smart contract governance to control the use of these functions.

If removing the functions or implementing the smart contract governance is not possible, Inspex suggests mitigating the risk of this issue by using a timelock mechanism to delay the changes for a reasonable amount of time (at least 24 hours).

5.5. Use of Upgradable Contract Design

ID	IDX-005
Target	APHCore APHPool
Category	General Smart Contract Vulnerability
CWE	CWE-284: Improper Access Control
Risk	**Severity: High** **Impact: High** The logic of affected contracts can be arbitrarily changed. This allows the proxy owner to perform malicious actions, e.g., stealing the users' funds anytime they want. **Likelihood: Medium** This action can be performed by the proxy owner without any restriction.
Status	Resolved * The FWX team has mitigated this issue by confirming to apply a timelock delay of at least 24 hours as the upgrade authority of the upgradable contracts, resulting in the delay of new implementation changes. The platform users should monitor the timelock for the execution of privileged actions such as contract upgrading and act accordingly.

5.5.1. Description

Smart contracts are designed to be used as agreements that cannot be changed forever. When a smart contract is upgraded, the agreement can be changed from what was previously agreed upon.

As these smart contracts are upgradable, the logic of them can be modified by the owner anytime, making the smart contracts untrustworthy

There are two contracts that have a function with the `initializer` modifier from the `Initializable` contract. The contracts that are designed to be compatible with the `Initializable` contract tend to be deployed as an implementation contract in an upgradable contract design.

APHCore.sol

```
constructor() initializer {}

function initialize(
    address _membershipAddress,
    address _forwAddress,
    address _routerAddress,
    address _wethAddress,
    address _wethHandlerAddress
```

```
) external initializer {
    manager = msg.sender;

    WEI_UNIT = 1018;
    WEI_PERCENT_UNIT = 1020;

    feeSpread = 10 ether;
    loanDuration = 28 days;
    advancedInterestDuration = 3 days;

    maxSwapSize = 1500 ether;

    routerAddress = _routerAddress;
    // routerAddress = 0x10ED43C718714eb63d5aA57B78B54704E256024E; // mainnet

    forwAddress = _forwAddress;

    membershipAddress = _membershipAddress;

    //AssetHandler_init_unchained parse 2 parameter _wethAddress ,_wethHandler
    __AssetHandler_init_unchained(_wethAddress, _wethHandlerAddress);

    emit SetRouterAddress(msg.sender, address(0), routerAddress);
    emit TransferManager(address(0), manager);
    emit SetLoanDuration(msg.sender, 0, loanDuration);
    emit SetFeeSpread(msg.sender, 0, feeSpread);
    emit SetAdvancedInterestDuration(msg.sender, 0, advancedInterestDuration);
    emit SetMembershipAddress(msg.sender, address(0), _membershipAddress);
}
```

APHPool.sol

```
constructor() initializer {}

function initialize(
    address _tokenAddress,
    address _coreAddress,
    address _membershipAddress,
    address _forwAddress,
    address _wethAddress,
    address _wethHandlerAddress,
    uint256 _blockTime
) external virtual initializer {
    require(_tokenAddress != address(0),
"APHPool/initialize/tokenAddress-zero-address");
    require(_coreAddress != address(0),
"APHPool/initialize/coreAddress-zero-address");
```

```
    require(_membershipAddress != address(0),
"APHPool/initialize/membership-zero-address");
    tokenAddress = _tokenAddress;
    coreAddress = _coreAddress;
    membershipAddress = _membershipAddress;
    manager = msg.sender;

    forwAddress = _forwAddress;
    interestVaultAddress = address(
        new InterestVault(tokenAddress, forwAddress, coreAddress, manager)
    );
    require(_blockTime != 0, "_blockTime cannot be zero");
    BLOCK_TIME = _blockTime;

    WEI_UNIT = 1018;
    WEI_PERCENT_UNIT = 1020;
    initialItpPrice = WEI_UNIT;
    initialIfpPrice = WEI_UNIT;
    lambda = 1 ether / 100;
    __AssetHandler_init_unchained(_wethAddress, _wethHandlerAddress);

    emit Initialize(manager, coreAddress, interestVaultAddress,
membershipAddress);
    emit TransferManager(address(0), manager);
}
```

5.5.2. Remediation

Inspex suggests deploying the contracts without the proxy pattern or any solution that can make the smart contracts upgradeable.

However, if upgradability is needed, Inspex suggests mitigating this issue by implementing a timelock mechanism with a sufficient length of time to delay the changes (at least 24 hours) on the proxy admin role. This allows the platform users to monitor the timelock and be notified of the potential changes being done on the smart contracts.

5.6. Design Flaw in Kill Switch Mechanism

ID	IDX-006
Target	PoolLending CoreBorrowing
Category	General Smart Contract Vulnerability
CWE	CWE-284: Improper Access Control
Risk	**Severity: High** **Impact: High** The platform users cannot withdraw their principal even in an emergency case due to the pausing of the withdrawal functionality. **Likelihood: Medium** It is likely that the centralized functions will be controlled by the authorized party since there is no mechanism to prevent it and there is a profit for doing so.
Status	**Acknowledged** The FWX team has acknowledged this issue since the team needs to ensure that if any incident occurs, all the functions can be paused to stop the functionalities of the platform, preventing additional damage. In addition, by allowing the platform users to withdraw while other functionalities are paused, the ratio of token in the pool will be changed, which can result in miscalculation which introduces another issue.

5.6.1. Description

The platform allows users to lend and borrow assets, providing them with the convenience of money management. However, there are several functions that can be controlled by the centralized party.

The following functions in the table show the high impact when it is not allowed to be executed by the platform user.

File	Function
PoolLending.sol (L: 84)	withdraw()
CoreBorrowing.sol (L: 48)	repay()
CoreBorrowing.sol (L: 156)	rollover()

For example, when a user wants to withdraw the principal, if the platform pauses the `withdraw()` function, the users will not be able to withdraw their funds, while it is possible for the platform to upgrade the contract and add dangerous functions that can affect the users' funds.

Furthermore, protection mechanisms such as timelock cannot save the users from this damage since kill switch mechanisms are usually not controlled in order to respond for emergency cases.

5.6.2. Remediation

Inspex suggests removing the ability to pause the affected functions to allow the users to retrieve their funds from the smart contract in an emergency case. This can be done by removing the `whenFuncNotPaused(msg.sig)` modifier from those functions, for example:

PoolLending.sol

```
function withdraw(uint256 nftId, uint256 withdrawAmount)
    external
    nonReentrant
    settleForwInterest
    returns (WithdrawResult memory)
{
```

5.7. Denial of Service in PoolLending

ID	IDX-007
Target	APHCore PoolBaseFunc PoolLending
Category	Advanced Smart Contract Vulnerability
CWE	CWE-755: Improper Handling of Exceptional Conditions
Risk	**Severity:** Medium **Impact:** High The users cannot execute core functions of the `PoolLending` contract, causing disruption of service and loss of reputation to the platform. The users will not be able to withdraw or deposit their funds due to this issue. **Likelihood:** Low It is unlikely that the platform will have insufficient funds for the vault.
Status	Resolved * The FWX team has mitigated this issue by confirming that the rewards will be distributed in a specific period (fixed total supply), which is during the period when the platform launches. Therefore, the total reward can be calculated beforehand and will be sufficient for all platform users.

5.7.1. Description

In the `PoolLending` contract, the core functions, such as the `withdraw()` function, have the `settleForwInterest` modifier that calls the `settleForwInterest()` function in the `APHCore` contract as shown below:

PoolLending.sol

```
function withdraw(uint256 nftId, uint256 withdrawAmount)
    external
    nonReentrant
    whenFuncNotPaused(msg.sig)
    settleForwInterest
    returns (WithdrawResult memory)
{
    nftId = _getUsableToken(msg.sender, nftId);
    WithdrawResult memory result = _withdraw(msg.sender, nftId, withdrawAmount);

    // transfer principal
```

```
    _transferOut(msg.sender, tokenAddress, result.principle);
    // transfer token interest
    _transferFromOut(
        interestVaultAddress,
        msg.sender,
        tokenAddress,
        result.tokenInterest + result.tokenInterestBonus
    );
    // transfer forw interest
    _transferFromOut(interestVaultAddress, msg.sender, forwAddress,
result.forwInterest);
    // transfer forw bonus from forwDis
    _transferFromOut(
        IAPHCore(coreAddress).forwDistributorAddress(),
        msg.sender,
        forwAddress,
        result.forwInterestBonus
    );
    return result;
}
```

PoolBaseFunc.sol

```
modifier settleForwInterest() {
    IAPHCore(coreAddress).settleForwInterest();
    _;
}
```

APHCore.sol

```
function settleForwInterest() external {
    require(poolToAsset[msg.sender] != address(0),
"APHCore/caller-is-not-pool");

    address interestVaultAddress = IAPHPool(msg.sender).interestVaultAddress();
    uint256 forwAmount = _settleForwInterest();
    _transferFromOut(forwDistributorAddress, interestVaultAddress, forwAddress,
forwAmount);

    emit SettleForwInterest(
        address(this),
        interestVaultAddress,
        forwDistributorAddress,
        forwAddress,
        forwAmount
    );
}
```

In line 59, the `_transferFromOut()` function can prevent the users from using the core functions in the `PoolLending` contract, since the function will be reverted if `forwAmount` is insufficient for transfer from the distributor.

5.7.2. Remediation

Inspex suggests implementing an emergency function for the users to withdraw their principal tokens in case the core functions are unusable, for example:

PoolLending.sol

```
function emergencyWithdraw(uint256 nftId)
    external
    nonReentrant
    returns (WithdrawResult memory)
{
    nftId = _getUsableToken(msg.sender, nftId);
    WithdrawResult memory result = _withdraw(msg.sender, nftId, type(uint256).max);

    // transfer principal
    _transferOut(msg.sender, tokenAddress, result.principle);
    // transfer token interest
    _transferFromOut(
        interestVaultAddress,
        msg.sender,
        tokenAddress,
        result.tokenInterest + result.tokenInterestBonus
    );
    // transfer forw interest
    _transferFromOut(interestVaultAddress, msg.sender, forwAddress, result.forwInterest);
    return result;
}
```

5.8. Denial of Service in Allowance Mechanism of InterestVault

ID	IDX-008
Target	InterestVault
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Medium **Impact:** Low The platform owner cannot add the allowance of the lending and FWX token address to the pool contract after the contract is deployed. Thus, the platform owner cannot change the token address anyway. **Likelihood:** High The `ownerApprove()` function will always be reverted.
Status	Resolved The FWX team has resolved this issue by replacing the `onlyOwner` modifier with the `onlyAddressTimelockManager` modifier and using `SafeERC20.safeIncreaseAllowance()` function instead of the `SafeERC20.safeApprove()` function in the `_ownerApprove()` function.

5.8.1. Description

Scenario 1: Inconsistency between Token Address and Its Allowance

The `setForwAddress()` and `setTokenAddress()` functions are used for changing the `forwAddress` and the `tokenAddress` respectively after being deployed and only manager (`onlyManager`) is allowed to call them as shown below in line 61 and 68:

InterestVault.sol

```
61 function setForwAddress(address _address) external onlyManager {
62     address oldAddress = forwAddress;
63     forwAddress = _address;
64
65     emit SetForwAddress(msg.sender, oldAddress, tokenAddress);
66 }
67
68 function setTokenAddress(address _address) external onlyManager {
69     address oldAddress = tokenAddress;
70     tokenAddress = _address;
71
72     emit SetTokenAddress(msg.sender, oldAddress, tokenAddress);
```

```
}
```

The `ownerApprove()` function can be used for adding allowance to any contract with `_pool` parameter. Therefore, this function allows the authorized party to give an allowance of `tokenAddress` and `forwAddress` from this contract to the `_pool` address.

With the `onlyOwner` modifier, only `_owner` can call the `ownerApprove()` function as shown in the following source code:

InterestVault.sol

```
function ownerApprove(address _pool) external onlyOwner {
    _ownerApprove(_pool);
}
```

InterestVault.sol

```
function _ownerApprove(address _pool) internal {
    uint256 approveAmount = type(uint256).max;
    IERC20(tokenAddress).safeApprove(_pool, approveAmount);
    IERC20(forwAddress).safeApprove(_pool, approveAmount);

    emit OwnerApprove(msg.sender, tokenAddress, forwAddress, approveAmount);
}

```

However, as the `InterestVault` contract is deployed by the `APHPool` contract as shown below in line 33, the owner of the `InterestVault` contract is the `APHPool` contract.

APHPool.sol

```
14 function initialize(
15     address _tokenAddress,
16     address _coreAddress,
17     address _membershipAddress,
18     address _forwAddress,
19     address _wethAddress,
20     address _wethHandlerAddress,
21     uint256 _blockTime
22 ) external virtual initializer {
23     require(_tokenAddress != address(0),
   "APHPool/initialize/tokenAddress-zero-address");
24     require(_coreAddress != address(0),
   "APHPool/initialize/coreAddress-zero-address");
25     require(_membershipAddress != address(0),
   "APHPool/initialize/membership-zero-address");
26     tokenAddress = _tokenAddress;
27     coreAddress = _coreAddress;
```

```
    membershipAddress = _membershipAddress;
    manager = msg.sender;

    forwAddress = _forwAddress;
    interestVaultAddress = address(
        new InterestVault(tokenAddress, forwAddress, coreAddress, manager)
    );
    require(_blockTime != 0, "_blockTime cannot be zero");
    BLOCK_TIME = _blockTime;

    WEI_UNIT = 1018;
    WEI_PERCENT_UNIT = 1020;
    initialItpPrice = WEI_UNIT;
    initialIfpPrice = WEI_UNIT;
    lambda = 1 ether / 100;
    __AssetHandler_init_unchained(_wethAddress, _wethHandlerAddress);

    emit Initialize(manager, coreAddress, interestVaultAddress, membershipAddress);
    emit TransferManager(address(0), manager);
}
```

From current implementation, the `APHPool` cannot call the `ownerApprove()` function of `InterestVault` contract. When the `manager` changes the FWX token via the `setForwAddress()` function or deposited token address via the `setTokenAddress()` function, they will not be able to add the allowance of new token to the `APHPool` contract.

Scenario 2: Multiple Approval at the Same Token Address

In the `_ownerApprove()` function, the `SafeERC20.safeApprove()` function is used for adding allowance.

The `SafeERC20.safeApprove()` function validates that the current allowance must be zero in line 57 as shown below:

SafeERC20Upgradeable.sol

```
41 /**
42     * @dev Deprecated. This function has issues similar to the ones found in
43     * {IERC20-approve}, and its usage is discouraged.
44     *
45     * Whenever possible, use {safeIncreaseAllowance} and
46     * {safeDecreaseAllowance} instead.
47     */
48 function safeApprove(
49     IERC20Upgradeable token,
50     address spender,
51     uint256 value
```

```
) internal {
    // safeApprove should only be called when setting an initial allowance,
    // or when resetting it to zero. To increase and decrease it, use
    // 'safeIncreaseAllowance' and 'safeDecreaseAllowance'
    require(
        (value == 0) || (token.allowance(address(this), spender) == 0),
        "SafeERC20: approve from non-zero to non-zero allowance"
    );
    _callOptionalReturn(token, abi.encodeWithSelector(token.approve.selector, spender, value));
}
```

Thus, in the case that the authorized party changes the address of either `forwAddress` or `tokenAddress`, whereas the other one is the same as before, the `ownerApprove()` function will always be reverted as there is one token address that already has the allowance for that pool address.

5.8.2. Remediation

Inspex suggests replacing the the `onlyOwner` modifier of `ownerApprove()` function to `onlyManager` modifier. For example:

InterestVault.sol

```
function ownerApprove(address _pool) external onlyManager {
    _ownerApprove(_pool);
}
```

Additionally, the `SafeERC20.safeApprove()` has already been deprecated and it can cause scenario 2 of this issue. Inspex suggests using `SafeERC20.safeIncreaseAllowance()` function instead of the `SafeERC20.safeApprove()` function, for example:

InterestVaultEvent.sol

```
event OwnerApprove(
    address indexed sender,
    address tokenAddress,
    address forwAddress,
    uint256 tokenApproveAmount,
    uint256 forwApproveAmount
);
```

InterestVault.sol

```
function _ownerApprove(address _pool, uint256 tokenApproveAmount, uint256 forwApproveAmount) internal {
    IERC20(tokenAddress).safeIncreaseAllowance(_pool, tokenApproveAmount);
    IERC20(forwAddress).safeIncreaseAllowance(_pool, forwApproveAmount);

```

```
    emit OwnerApprove(msg.sender, tokenAddress, forwAddress,
tokenApproveAmount, forwApproveAmount);
}
```

5.9. Position Liquidation Avoidance

ID	IDX-009
Target	CoreBorrowing
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Medium **Impact:** Medium The loan position's owner with the native token as collateral can decide to make the position liquidatable or not. This means the loan position can be avoided from the liquidation, resulting in bad debt for the platform. Since the liquidation is already avoided, the position's owner can repay the debt and take the collateral back normally. Therefore, the platform will lose the incentive from liquidation (liquidation bounty reward). **Likelihood:** Medium It is likely that the loan position's owner with native tokens as collateral will make his or her position unable to be liquidated since it guarantees that the collateral will be saved from liquidation, which can be taken back anytime by repaying the loan. However, this scenario will happen only when the liquidation position is not in bad debt, so there is `leftOverCollateral` from the liquidation swap to trigger the contract flow hijacking.
Status	Resolved The FWX team has resolved this issue as suggested by modifying the `liquidate()` function to transfer wrapped native tokens instead of native tokens.

5.9.1. Description

When the collateral asset price is under collateral threshold, the platform allows anyone to liquidate that position immediately to prevent the bad debt for the platform, which can be performed through the `liquidate()` function.

At line 200, the `leftOverCollateral` asset, if exists, will be transferred back to the owner of position through the `_trasnferOut()` function.

CoreBorrowing.sol

```
178 function liquidate(uint256 loanId, uint256 nftId)
179     external
180     whenFuncNotPaused(msg.sig)
181     nonReentrant
182     returns (
183         uint256 repayBorrow,
184         uint256 repayInterest,
```

```
        uint256 bountyReward,
        uint256 leftOverCollateral
    )
{
    Loan storage loan = loans[nftId][loanId];
    (repayBorrow, repayInterest, bountyReward, leftOverCollateral) =
_liquidate(loanId, nftId);

    _transferOut(assetToPool[loan.borrowTokenAddress], loan.borrowTokenAddress,
repayBorrow);
    _transferOut(
        IAPHPool(assetToPool[loan.borrowTokenAddress]).interestVaultAddress(),
        loan.borrowTokenAddress,
        repayInterest
    );

    _transferOut(msg.sender, loan.collateralTokenAddress, bountyReward);
    _transferOut(_getTokenOwnership(nftId), loan.collateralTokenAddress,
leftOverCollateral);
}
```

If the collateral asset is a wrapped native token, it will be unwrapped to the native token and transferred to the target address, in this case it is the owner of the NFT position which can be EOA or contract address.

AssetHandlerUpgradeable.sol

```
function _transferOut(
    address to,
    address token,
    uint256 amount
) internal {
    if (amount == 0) {
        return;
    }
    if (token == wethAddress) {
        IWethERC20(wethAddress).safeTransfer(wethHandler, amount);
        IWethHandler(payable(wethHandler)).withdrawETH(to, amount);
    } else {
        IERC20(token).safeTransfer(to, amount);
    }
}
```

After being unwrapped successfully, it will transfer the native token to the target as in line 14. When calling the target address with `call()`, it will give the execution flow to the target address if it is a contract.

WETHHandler.sol

```
function withdrawETH(address to, uint256 amount) external {
    IWeth(wethAddress).withdraw(amount);
    (bool success, ) = to.call{value: amount}(new bytes(0));
    require(success, "WETHHandler/withdraw-failed-1");
}
```

As a result, if the position owner is the contract with a revert mechanism every time that the native is transferred to the position (`liquidate()` is triggered), it will not be able to be liquidated due to the force reverting.

5.9.2. Remediation

Inspex suggests transferring wrapped native tokens instead of native tokens. This also matches to the business design as consulted in which the platform allows using wrapped native tokens only. For example:

CoreBorrowing.sol

```
// SPDX-License-Identifier: GPL-3.0

pragma solidity 0.8.14;

import "./event/CoreBorrowingEvent.sol";
import "./CoreBaseFunc.sol";
import "../../externalContract/openzeppelin/non-upgradeable/IERC20.sol";
```

CoreBorrowing.sol

```
function liquidate(uint256 loanId, uint256 nftId)
    external
    whenFuncNotPaused(msg.sig)
    nonReentrant
    returns (
        uint256 repayBorrow,
        uint256 repayInterest,
        uint256 bountyReward,
        uint256 leftOverCollateral
    )
{
    Loan storage loan = loans[nftId][loanId];
    (repayBorrow, repayInterest, bountyReward, leftOverCollateral) =
_liquidate(loanId, nftId);
IERC20(loan.borrowTokenAddress).safeTransfer(assetToPool[loan.borrowTokenAddress], repayBorrow);

IERC20(loan.borrowTokenAddress).safeTransfer(IAPHPool(assetToPool[loan.borrowTokenAddress]).interestVaultAddress(), repayInterest);
```

```
    IERC20(loan.collateralTokenAddress).safeTransfer(msg.sender, bountyReward);
    IERC20(loan.collateralTokenAddress).safeTransfer(_getTokenOwnership(nftId),
leftOverCollateral);
}
```

5.10. Design Flaw in Liquidation Mechanism

ID	IDX-010
Target	CoreBorrowing
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Medium **Impact:** High The loan position cannot be closed in the liquidation process if the collateral value is insufficient for closing the loan. As a result, the platform incurs an irrecoverable debt. **Likelihood:** Low It is unlikely that the collateral value will be insufficient for the loan position, since there is a gap between the maximum loan to value and the value of the collateral. If the value of the collateral drops below the liquidation point, the users are incentivized to liquidate the position before it reaches bad debt.
Status	Resolved The FWX team has resolved this issue as suggested by modifying the liquidation flow to support the bad debt scenario. When the collateral is insufficient to cover the loan principal, the platform will record a `loss` amount, which will be used to distribute the loss to the users who lend their liquidity before the bad debt liquidation occurs.

5.10.1. Description

In the `CoreBorrowing` contract, the `_liquidate()` function calls the `_repay()` function to repay for the loan position as shown in line 596.

CoreBorrowing.sol

```
550 function _liquidate(uint256 loanId, uint256 nftId)
551     internal
552     returns (
553         uint256 repayBorrow,
554         uint256 repayInterest,
555         uint256 bountyReward,
556         uint256 leftOverCollateral
557     )
558 {
559     Loan storage loan = loans[nftId][loanId];
560     LoanConfig storage loanConfig = loanConfigs[loan.borrowTokenAddress][
561         loan.collateralTokenAddress
562     ];
```

```
    // rate, precision, maxSwappable (collaToken => borrowToken)
    uint256[] memory numberArray = new uint256[](3);
    require(loanExts[nftId][loanId].active == true, "CoreBorrowing/loan-is-closed");

    _settleBorrowInterest(loan);

    (numberArray[0], numberArray[1]) = _queryRate(
        loan.collateralTokenAddress,
        loan.borrowTokenAddress
    );

    // rollover if loan is overdue
    if (block.timestamp > loan.rolloverTimestamp) {
        (, bountyReward) = _rollover(loanId, nftId, msg.sender);
    }

    // liquidate
    if (
        _isLoanLTVExceedTargetLTV(
            loan.borrowAmount,
            loan.collateralAmount,
            MathUpgradeable.max(loan.interestOwed, loan.minInterest),
            loanConfig.liquidationLTV,
            numberArray[0],
            numberArray[1]
        )
    ) {
        (uint256 collateralTokenAmountUsed, uint256 borrowTokenAmountSwap) = _liquidationSwap(
            loan
        );

        leftOverCollateral = loan.collateralAmount - collateralTokenAmountUsed;

        (repayBorrow, repayInterest, ) = _repay(loanId, nftId, borrowTokenAmountSwap, false);

        if (loanExts[nftId][loanId].active == true) {
            // TODO (future work): handle with ciritical condition, this part must add pool subsidisation for pool loss
            // Ciritical condition, protocol loss transfer int or sth else to pool
        } else {
            bountyReward += (leftOverCollateral * loanConfig.bountyFeeRate) / WEI_PERCENT_UNIT;
            leftOverCollateral -=
```

```
                (leftOverCollateral * loanConfig.bountyFeeRate) /
                WEI_PERCENT_UNIT;
        }

        emit Liquidate(
            msg.sender,
            nftId,
            loanId,
            msg.sender,
            numberArray[0],
            borrowTokenAmountSwap,
            bountyReward,
            loan.collateralTokenAddress,
            leftOverCollateral
        );
    }
}
```

If the whole debt is repaid, the loan active status will be set to `false` as shown in line 388.

CoreBorrowing.sol

```
332 function _repay(
333     uint256 loanId,
334     uint256 nftId,
335     uint256 repayAmount,
336     bool isOnlyInterest
337 )
338     internal
339     returns (
340         uint256 borrowPaid,
341         uint256 interestPaid,
342         bool isLoanClosed
343     )
344 {
345     Loan storage loan = loans[nftId][loanId];
346     PoolStat storage poolStat =
poolStats[assetToPool[loan.borrowTokenAddress]];
347
348     require(loanExts[nftId][loanId].active == true,
"CoreBorrowing/loan-is-closed");
349
350     _settleBorrowInterest(loan);
351
352     uint256 collateralAmountWithdraw = 0;
353
354     // pay only interest
355     if (isOnlyInterest || repayAmount <= loan.interestOwed) {
```

```
        interestPaid = MathUpgradeable.min(repayAmount, loan.interestOwed);
        loan.interestOwed -= interestPaid;
        loan.interestPaid += interestPaid;

        if (loan.minInterest > interestPaid) {
            loan.minInterest -= interestPaid;
        } else {
            loan.minInterest = 0;
        }

        poolStat.totalInterestPaid += interestPaid;
    } else {
        interestPaid = MathUpgradeable.max(loan.minInterest, loan.interestOwed);
        if (repayAmount >= (loan.borrowAmount + interestPaid)) {
            // close loan
            poolStat.totalInterestPaid += interestPaid;
            poolStat.totalBorrowAmount -= loan.borrowAmount;
            poolStat.borrowInterestOwedPerDay -= loan.owedPerDay;

            collateralAmountWithdraw = loan.collateralAmount;

            totalCollateralHold[loan.collateralTokenAddress] -= collateralAmountWithdraw;

            borrowPaid = loan.borrowAmount;
            loan.minInterest = 0;
            loan.interestOwed = 0;
            loan.owedPerDay = 0;
            loan.borrowAmount = 0;
            loan.collateralAmount = 0;
            loan.interestPaid += interestPaid;

            isLoanClosed = true;
            loanExts[nftId][loanId].active = false;
        } else {
            // pay int and some of principal
            uint256 oldBorrowAmount = loan.borrowAmount;

            interestPaid = MathUpgradeable.min(interestPaid, loan.interestOwed);
            loan.interestPaid += interestPaid;

            borrowPaid = MathUpgradeable.min(repayAmount - interestPaid, loan.borrowAmount);
            loan.borrowAmount -= borrowPaid;

```

```
            poolStat.borrowInterestOwedPerDay -= loan.owedPerDay;

            // set new owedPerDat
            loan.owedPerDay = (loan.owedPerDay * loan.borrowAmount) / oldBorrowAmount;
            poolStat.borrowInterestOwedPerDay += loan.owedPerDay;

            if (loan.minInterest > loan.interestOwed) {
                loan.minInterest -= interestPaid;
            } else {
                loan.minInterest = 0;
            }

            loan.interestOwed -= interestPaid;
            poolStat.totalInterestPaid += interestPaid;
            poolStat.totalBorrowAmount -= borrowPaid;
        }
    }

IInterestVault(IAPHPool(assetToPool[loan.borrowTokenAddress]).interestVaultAddress())
        .settleInterest(
            (interestPaid * (WEI_PERCENT_UNIT - feeSpread)) / WEI_PERCENT_UNIT,
            (interestPaid * feeSpread) / WEI_PERCENT_UNIT,
            0
        );

    emit Repay(
        msg.sender,
        nftId,
        loanId,
        collateralAmountWithdraw > 0,
        borrowPaid,
        interestPaid,
        collateralAmountWithdraw
    );
}
```

However, if the collateral token is insufficient to cover all of the debt, the state of loan position will stay as active, and since all of the collateral is already used, the position can never be liquidated by others in the future, resulting in an irrecoverable debt.

5.10.2. Remediation

Inspex suggests redesigning the liquidation business flow by allowing the liquidator to provide the borrowed asset for the debt repayment and rewarding the liquidator with the collateral of the position (the liquidator buys the collateral asset at a cheaper price compared to the market price).

5.11. Transaction Ordering Dependence in _liquidationSwap() Function

ID	IDX-011
Target	CoreBorrowing
Category	Advanced Smart Contract Vulnerability
CWE	CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')
Risk	**Severity:** Medium **Impact:** Medium The front running attack can be performed, resulting in a bad swapping rate and might cause bad debt to the platform. **Likelihood:** Medium It is likely to perform the attack since the attacker can get profit from the price impact.
Status	Resolved * The FWX team has resolved this issue by calculating the expected amount out with the token price fetched from the price oracles, and setting it to the `amountOutMin` parameter while calling the swapping function. However, fetching the token price from the oracle might not be accurate at the time. Using this price for swapping tokens will still have a price impact and the transaction can be reverted if the token price from the oracle is higher than the on-chain token price, so the price slippage should also be adjusted. With the revert of the liquidation process, it means the bad debt could occur to the platform.

5.11.1. Description

In the `CoreBorrowing` contract, the `_liquidate()` function calls the `_liquidationSwap()` function for swapping the collateral token to borrowed token and then repay the debt as shown below:

CoreBorrowing.sol

```
function _liquidationSwap(Loan storage loan)
    internal
    returns (uint256 collateralTokenAmountUsed, uint256 borrowTokenAmountSwap)
{
    address[] memory path_data = new address[](2);
    path_data[0] = loan.collateralTokenAddress;
    path_data[1] = loan.borrowTokenAddress;
    uint256[] memory amounts;

    uint256 amountOut =
IRouter(routerAddress).getAmountsOut(loan.collateralAmount, path_data)[
        1
```

```
    ];
    if (
        amountOut > loan.borrowAmount + MathUpgradeable.max(loan.interestOwed,
loan.minInterest)
    ) {
        amountOut =
            loan.borrowAmount +
            MathUpgradeable.max(loan.interestOwed, loan.minInterest);

        // Normal condition, leftover collateral is exists
        amounts = IRouter(routerAddress).swapTokensForExactTokens(
            amountOut, //           // amountOut
            loan.collateralAmount, //    // amountInMax
            path_data,
            address(this),
            1 hours + block.timestamp
        );
    } else {
        amounts = IRouter(routerAddress).swapExactTokensForTokens(
            loan.collateralAmount, //    // amountIn
            0, //                        // amountOutMin
            path_data,
            address(this),
            1 hours + block.timestamp
        );
    }
    collateralTokenAmountUsed = amounts[0];
    borrowTokenAmountSwap = amounts[amounts.length - 1];
}
```

However, as seen in the source code above, if the price is inflated, the `IRouter(routerAddress).swapExactTokensForTokens()` function is called with the `amountOutMin` value equal to 0. Therefore, the front running attack can be performed, resulting in a bad swapping rate and a lower bounty.

5.11.2. Remediation

Inspex suggests redesigning the liquidate mechanism as suggested in **IDX-010**.

If redesigning the mechanism is not possible, the price impact can be reduced by calculating the expected amount out with the token price fetched from the price oracles, and setting it to the `amountOutMin` parameter while calling the `IRouter(routerAddress).swapTokensForExactTokens()` and the `IRouter(routerAddress).swapExactTokensForTokens()` functions as shown in the following example at lines 631 and 649:

Note: Fetching the token price from the oracle might not be accurate at the time. Using this price for swapping tokens will still have a price impact and the transaction can be reverted if the token price from the oracle is higher than the on-chain token price, so the price slippage should also be adjusted. With the revert of the liquidation process, it means the bad debt will occur to the platform.

CoreBorrowing.sol

```
622 function _liquidationSwap(Loan storage loan)
623     internal
624     returns (uint256 collateralTokenAmountUsed, uint256 borrowTokenAmountSwap)
625 {
626     address[] memory path_data = new address[](2);
627     path_data[0] = loan.collateralTokenAddress;
628     path_data[1] = loan.borrowTokenAddress;
629     uint256[] memory amounts;
630 
631     uint256 amountOut = calculateAmountOutFromOracle(loan.collateralAmount,
path_data);
632 
633     if (
634         amountOut > loan.borrowAmount + MathUpgradeable.max(loan.interestOwed,
loan.minInterest)
635     ) {
636         amountOut =
637             loan.borrowAmount +
638             MathUpgradeable.max(loan.interestOwed, loan.minInterest);
639 
640         // Normal condition, leftover collateral is exists
641         amounts = IRouter(routerAddress).swapTokensForExactTokens(
642             amountOut, //          // amountOut
643             loan.collateralAmount, //    // amountInMax
644             path_data,
645             address(this),
646             1 hours + block.timestamp
647         );
648     } else {
649         uint256 amountOutMin =
```

```
calculateAmountOutMinFromOracle(loan.collateralAmount, path_data);
        amounts = IRouter(routerAddress).swapExactTokensForTokens(
            loan.collateralAmount, //   // amountIn
            amountOutMin, //                       // amountOutMin
            path_data,
            address(this),
            1 hours + block.timestamp
        );
    }
    collateralTokenAmountUsed = amounts[0];
    borrowTokenAmountSwap = amounts[amounts.length - 1];
}
```

5.12. Unnecessary Rate Validation in _queryRateUSD() Function

ID	IDX-012
Target	PriceFeeds_BSC
Category	Advanced Smart Contract Vulnerability
CWE	CWE-1164: Irrelevant Code
Risk	**Severity:** Low **Impact:** Medium The functions that call the `queryRate()` function, e.g., `APHCore.borrow()`, and `APHCore.liquidate()`, will be reverted when the `rate` is more than `2^128`. This causes the main functionalities of the platform to be unusable. **Likelihood:** Low In general, the decimal of the rate from the price oracle contract is `8`. As a result, the price of the token in $USD rate is more than `3.4e30`, making it very unlikely for this issue to happen.
Status	Resolved The FWX team has resolved this issue as suggested by removing the rate validation that checks the `rate` state in.

5.12.1. Description

The `queryRate()` function is used to query the rate of the `sourceToken` and the `destToken` from the price oracle contract via the `_queryRate()` function as shown below in line 38.

PriceFeed.sol

```
32 function queryRate(address sourceToken, address destToken)
33     public
34     view
35     returns (uint256 rate, uint256 precision)
36 {
37     require(!globalPricingPaused, "PriceFeed/pricing-is-paused");
38     return _queryRate(sourceToken, destToken);
39 }
```

In the `_queryRate()` function, the USD rates are queried by the `_queryRateUSD()` function as shown below in lines 136-137. The rate of `sourceToken` per `destToken` is then calculated in line 139 by dividing the `sourceRate` with `destRate`.

PriceFeed.sol

```
function _queryRate(address sourceToken, address destToken)
    internal
    view
    returns (uint256 rate, uint256 precision)
{
    if (sourceToken != destToken) {
        uint256 sourceRate = _queryRateUSD(sourceToken);
        uint256 destRate = _queryRateUSD(destToken);

        rate = (sourceRate * WEI_PRECISION) / destRate;

        precision = _getDecimalPrecision(sourceToken, destToken);
    } else {
        rate = WEI_PRECISION;
        precision = WEI_PRECISION;
    }
}
```

In the `_queryRateUSD()` function, the code is validating the `rate` that is queried from the price oracle to ensure the `rate` is less than `2^128` as shown at line 152.

PriceFeed.sol

```
148 function _queryRateUSD(address token) internal view returns (uint256 rate) {
149     require(pricesFeeds[token] != address(0), "PriceFeed/unsupported-address");
150     AggregatorV2V3Interface _Feed =
    AggregatorV2V3Interface(pricesFeeds[token]);
151     rate = uint256(_Feed.latestAnswer());
152     require(rate != 0 && (rate >> 128) == 0, "PriceFeed/price-error");
153 }
```

Since the `queryRateUSD()` function is called by multiple functions e.g. `APHCore.borrow()` and `APHCore.liquidate()`, they will be reverted when the `rate` is more than `2^128`.

However, the possibility of the concern scenario is pretty low. Since the decimal of the `rate` from the price oracle contract is `8` in general. This scenario requires the price of the token in $USD to be worth more than `2^128` $USD divided by `10^8` (decimals), or `~3.4e30` $USD (34,000,000,000).

5.12.2. Remediation

Inspex suggests removing the rate validation in line 152 that check the `rate` must be more than `2^128`, for example:

PriceFeed.sol

```
148 function _queryRateUSD(address token) internal view returns (uint256 rate) {
149     require(pricesFeeds[token] != address(0), "PriceFeed/unsupported-address");
150     AggregatorV2V3Interface _Feed =
AggregatorV2V3Interface(pricesFeeds[token]);
151     rate = uint256(_Feed.latestAnswer());
152     require(rate != 0, "PriceFeed/price-error");
153 }
```

5.13. Incorrect State Variable Setting in setMembershipAddress() Function

ID	IDX-013
Target	PoolSetting
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Low **Impact:** Medium The authorized party (`onlyManager` modifier) cannot configure the NFT address (`membershipAddress` state variable) for the pool contracts, preventing the membership contract from being updated, and unintentionally changing the `forwAddress`, temporarily disrupting the service of the platform. However, the incorrect configuration can be changed back to the correct one by executing this function with the correct parameter again. **Likelihood:** Low It is unlikely that the authorized party will call the `setMembershipAddress()` function to change the `membershipAddress` state variable as intended or `forwAddress` state variable according to the code. This is because it is not mandatory to do so in the business design, and the state has been set during the contract initialization.
Status	Resolved The FWX team has resolved this issue as suggested by modifying the state variable to match the intention of the function.

5.13.1. Description

The `PoolSetting` contract provides the function to let the authorized party, as specified in the `onlyManager` modifier, to configure state variables. For example, the `setMembershipAddress()` function can be used to set the membership contract for a specific pool.

PoolSetting.sol

```
function setMembershipAddress(address _address) external onlyManager {
    address oldAddress = forwAddress;
    forwAddress = _address;

    emit SetMembershipAddress(msg.sender, oldAddress, _address);
}
```

However, the state being set in the `setMembershipAddress()` function is `forwAddress` address instead. This means the new membership contract for that pool cannot be set again, and the `forwAddress` will be unintentionally changed, temporarily disrupting the service of the platform.

5.13.2. Remediation

Inspex suggests modifying the state variable to match the intention of the function, which should be the `membershipAddress` state variable.

PoolSetting.sol

```
function setMembershipAddress(address _address) external onlyManager {
    address oldAddress = membershipAddress;
    membershipAddress = _address;

    emit SetMembershipAddress(msg.sender, oldAddress, _address);
}
```

5.14. Improper Interest Reward Distribution

ID	IDX-014
Target	PoolLending
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Low **Impact:** Low The interest reward distribution for the platform users will be less than the actual since the interest will be only accrual when the borrower repays their loan. Resulting in unfair interest reward distribution. **Likelihood:** Medium The interest reward distribution will be miscalculated when the principal token is changed without updating the platform interest.
Status	**Acknowledged** The FWX team has acknowledged and accepted the issue since the user's reward interest will be slightly impacted.

5.14.1. Description

In the `PoolLending` contract, the users can call the `deposit()` function to lending liquidity to the platform and get the interest as a reward as shown below:

PoolLending.sol

```
function _deposit(
    address receiver,
    uint256 nftId,
    uint256 depositAmount
)
    internal
    returns (
        uint256 pMintAmount,
        uint256 itpMintAmount,
        uint256 ifpMintAmount
    )
{
    require(depositAmount > 0, "PoolLending/deposit-amount-is-zero");

    Lend storage lend = lenders[nftId];

    uint256 itpPrice = _getInterestTokenPrice();
```

```
    uint256 ifpPrice = _getInterestForwPrice();

    //mint ip,itp,ifp
    pMintAmount = _mintPToken(receiver, nftId, depositAmount);
    itpMintAmount = _mintItpToken(
        receiver,
        nftId,
        ((depositAmount * WEI_UNIT) / itpPrice),
        itpPrice
    );
    ifpMintAmount = _mintIfpToken(
        receiver,
        nftId,
        ((depositAmount * WEI_UNIT) / ifpPrice),
        ifpPrice
    );

    lend.updatedTimestamp = uint64(block.timestamp);

    emit Deposit(receiver, nftId, depositAmount, pMintAmount, itpMintAmount, ifpMintAmount);
}
```

The interest reward is calculated in the `_getInterestTokenPrice()` function by using `claimableTokenInterest` as the factor as shown below in line 150:

PoolBaseFunc.sol

```
144 function _getInterestTokenPrice() internal view returns (uint256) {
145     if (itpTokenTotalSupply == 0) {
146         return initialItpPrice;
147     } else {
148         return
149             ((pTokenTotalSupply +
150                 IInterestVault(interestVaultAddress).claimableTokenInterest()) * WEI_UNIT) /
151             itpTokenTotalSupply;
152     }
153 }
```

The `claimableTokenInterest` will only update when user repay the debt back to the pool in the `_settleInterest()` function as shown below in line 152:

InterestVault.sol

```
147 function _settleInterest(
148     uint256 _claimableTokenInterest,
149     uint256 _heldTokenInterest,
```

```
    uint256 _claimableForwInterest
) internal {
    claimableTokenInterest += _claimableTokenInterest;
    heldTokenInterest += _heldTokenInterest;
    claimableForwInterest += _claimableForwInterest;

    emit SettleInterest(
        msg.sender,
        claimableTokenInterest,
        heldTokenInterest,
        claimableForwInterest
    );
}
```

This results in unfair calculation in the interest reward distribution for users. The following example scenario shows how the interest reward is being distributed unfairly:

1. A - `deposit()` at block 0
2. B - `borrow()` at block 0
3. C - `deposit()` at block 100
4. B - `repay()` at block 100 after C has deposited, the `claimableTokenInterest` is increased.
5. A and C gain equal share of the interest paid from B

With the current design, C will gain profit suddenly without lending liquidity to the platform.

5.14.2. Remediation

Inspex suggests redesigning the interest reward distribution mechanism by accruing interest over time. When the platform users claim their interests, the time of liquidity provision should be one of factors to calculate the interest amount, incentivizing them to lend the liquidity to the platform. Therefore, the lenders will gain interest over time instead of gaining the share of the interest at the exact time when the interest is paid to the platform.

5.15. Smart Contract with Unpublished Source Code

ID	IDX-015
Target	All contracts
Category	General Smart Contract Vulnerability
CWE	CWE-1006: Bad Coding Practices
Risk	**Severity:** Low **Impact:** Medium The logic of the smart contract may not align with the user's understanding, causing undesired actions to be taken when the user interacts with the smart contract. **Likelihood:** Low The possibility for the users to misunderstand the functionalities of the contract is not very high with the help of the documentation and user interface.
Status	**Acknowledged** The FWX team has acknowledged this issue since the platform is under development, so they are not ready to publish the source code yet.

5.15.1. Description

The smart contract source code is not publicly published, so the users will not be able to easily verify the correctness of the functionalities and the logic of the smart contract by themselves. Therefore, it is possible that the user's understanding of the smart contract does not align with the actual implementation, leading to undesired actions on interacting with the smart contract.

5.15.2. Remediation

Inspex suggests publishing the contract source code through a public code repository or verifying the smart contract source code on the blockchain explorer so that the users can easily read and understand the logic of the smart contract by themselves.

5.16. Missing Input Validation in setRankInfo() Function

ID	IDX-016
Target	StakePool
Category	Advanced Smart Contract Vulnerability
CWE	CWE-20: Improper Input Validation
Risk	**Severity:** Very Low **Impact:** Low The `setRankInfo()` function can be reverted due to the array being out of bounds in looping. This is because there is no input validation for `_maxLTVBonus` array. **Likelihood:** Low It is unlikely that the execution of `setRankInfo()` function will fail since the configuration for each rank is designed to have its individual value, so the length of all the arrays could be the same.
Status	**Resolved** The FWX team has resolved this issue as suggested by adding an input validation for `_maxLTVBonus` array length.

5.16.1. Description

The `setRankInfo()` function in the `StakePool` contract allows the authorized party (`onlyOwner`) to set the benefit by staking $FORW.

At line 79, there is an input validation check to verify the input of the array.

StakePool.sol

```
72 function setRankInfo(
73     uint256[] memory _interestBonusLending,
74     uint256[] memory _forwardBonusLending,
75     uint256[] memory _minimumstakeAmount,
76     uint256[] memory _maxLTVBonus,
77     uint256[] memory _tradingFee
78 ) external onlyOwner {
79     require(
80         _interestBonusLending.length == _forwardBonusLending.length &&
81             _forwardBonusLending.length == _minimumstakeAmount.length &&
82             _forwardBonusLending.length == _tradingFee.length,
83         "input-does-not-have-same-length"
84     );
85 
86     for (uint8 i = 0; i < _interestBonusLending.length; i++) {
```

```
        RankInfo memory rankInfo = RankInfo(
            _interestBonusLending[i],
            _forwardBonusLending[i],
            _minimumstakeAmount[i],
            _maxLTVBonus[i],
            _tradingFee[i]
        );
        rankInfos[i] = rankInfo;
    }
    rankLen = uint8(_interestBonusLending.length);
}
```

However, there is no array length check for `_maxLTVBonus` array.

5.16.2. Remediation

Inspex suggests adding an input validation for `_maxLTVBonus` array length at line 82.

StakePool.sol

```
72 function setRankInfo(
73     uint256[] memory _interestBonusLending,
74     uint256[] memory _forwardBonusLending,
75     uint256[] memory _minimumstakeAmount,
76     uint256[] memory _maxLTVBonus,
77     uint256[] memory _tradingFee
78 ) external onlyOwner {
79     require(
80         _interestBonusLending.length == _forwardBonusLending.length &&
81             _forwardBonusLending.length == _minimumstakeAmount.length &&
82             _forwardBonusLending.length == _maxLTVBonus.length &&
83             _forwardBonusLending.length == _tradingFee.length,
84         "input-does-not-have-same-length"
85     );
86
87     for (uint8 i = 0; i < _interestBonusLending.length; i++) {
88         RankInfo memory rankInfo = RankInfo(
89             _interestBonusLending[i],
90             _forwardBonusLending[i],
91             _minimumstakeAmount[i],
92             _maxLTVBonus[i],
93             _tradingFee[i]
94         );
95         rankInfos[i] = rankInfo;
96     }
97     rankLen = uint8(_interestBonusLending.length);
98 }
```

5.17. Insufficient Logging for Privileged Functions

ID	IDX-017
Target	Membership StakePool Vault
Category	General Smart Contract Vulnerability
CWE	CWE-778: Insufficient Logging
Risk	**Severity:** Very Low **Impact:** Low Privileged functions' executions cannot be monitored easily by the users. **Likelihood:** Low It is not likely that the execution of the privileged functions will be a malicious action.
Status	Resolved The FWX team has resolved this issue as suggested by emitting events for the execution of privileged functions.

5.17.1. Description

Privileged functions that are executable by the controlling parties are not logged properly by emitting events. Without events, it is not easy for the public to monitor the execution of those privileged functions, allowing the controlling parties to perform actions that cause big impacts on the platform.

For example, the owner can set the new pool by executing the `setNewPool()` function in the `Membership` contract, and no events are emitted.

The privileged functions without sufficient logging are as follows:

File	Contract	Function
Membership.sol (L:34)	Membership	setNewPool()
Membership.sol (L:43)	Membership	setBaseURI()
StakePool.sol (L:37)	StakePool	setNextPool()
StakePool.sol (L:44)	StakePool	setSettleInterval()
StakePool.sol (L:51)	StakePool	setSettlePeriod()
StakePool.sol (L:72)	StakePool	setRankInfo()

Vault.sol (L:19)	Vault	ownerApprove()
Vault.sol (L:27)	Vault	approveInterestVault()

5.17.2. Remediation

Inspex suggests emitting events for the execution of privileged functions, for example:

Membership.sol

```
event SetNewPool(address newPool);
function setNewPool(address newPool) external onlyOwner whenNotPaused {
    currentPool = newPool;
    _poolIndex[newPool] = _poolList.length;
    _poolList.push(newPool);
    emit SetNewPool(newPool);
}
```

5.18. Outdated Compiler Version

ID	IDX-018
Target	All contracts
Category	General Smart Contract Vulnerability
CWE	CWE-1104: Use of Unmaintained Third Party Components
Risk	**Severity:** Very Low **Impact:** Low From the list of known Solidity bugs, direct impact cannot be caused from those bugs themselves. **Likelihood:** Low From the list of known Solidity bugs, it is very unlikely that those bugs would affect these smart contracts.
Status	**Acknowledged** The FWX team has acknowledged this issue. The current smart contract applies the Solidity compiler version 0.8.15. However, the list of known bugs does not affect the contracts in the scope, according to https://docs.soliditylang.org/en/v0.8.17/bugs.html.

5.18.1. Description

The Solidity compiler versions specified in the smart contracts were outdated. These versions have publicly known inherent bugs that may potentially be used to cause damage to the smart contracts or the users of the smart contracts.

APHCore.sol

```
pragma solidity 0.8.14;
```

The following table contains all contracts which the outdated compiler declares.

File	Version
CoreBorrowingEvent.sol (L:3)	0.8.14
CoreEvent.sol (L:3)	0.8.14
CoreFutureTradingEvent.sol (L:3)	0.8.14
CoreSettingEvent.sol (L:3)	0.8.14

APHCore.sol (L:3)	0.8.14
APHCoreProxy.sol (L:3)	0.8.14
CoreBase.sol (L:3)	0.8.14
CoreBaseFunc.sol (L:3)	0.8.14
CoreBorrowing.sol (L:3)	0.8.14
CoreFutureTrading.sol (L:3)	0.8.14
CoreSetting.sol (L:3)	0.8.14
Timelock.sol (L:3)	0.8.14
Membership.sol (L:3)	0.8.14
InterestVaultEvent.sol (L:3)	0.8.14
PoolLendingEvent.sol (L:3)	0.8.14
PoolSettingEvent.sol (L:3)	0.8.14
APHPool.sol (L:3)	0.8.14
APHPoolProxy.sol (L:3)	0.8.14
InterestVault.sol (L:3)	0.8.14
PoolBase.sol (L:3)	0.8.14
PoolBaseFunc.sol (L:3)	0.8.14
PoolBorrowing.sol (L:3)	0.8.14
PoolLending.sol (L:3)	0.8.14
PoolSetting.sol (L:3)	0.8.14
PoolToken.sol (L:3)	0.8.14
StakePool.sol (L:3)	0.8.14
StakePoolBase.sol (L:3)	0.8.14
PriceFeeds.sol (L:3)	0.8.14
ProxyAdmin.sol (L:3)	0.8.14
TransparentProxy.sol (L:3)	0.8.14

Vault.sol (L:3)	0.8.14
WETHHandler.sol (L:3)	0.8.14

5.18.2. Remediation

Inspex suggests upgrading the Solidity compiler to the latest stable version.

During the audit activity, the latest stable version of Solidity compiler in major 0.8 is v0.8.17 (https://github.com/ethereum/solidity/releases). For example:

APHCore.sol

```
pragma solidity 0.8.17;
```

5.19. Incorrect Logging for _withdraw() Function

ID	IDX-019
Target	PoolLending
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Info **Impact: None** **Likelihood: None**
Status	No Security Impact The FWX team has acknowledged this issue since the emitting event is designed to meet the business requirement.

5.19.1. Description

The `_withdraw()` function emits the `Withdraw` event to indicate the withdrawn token amount, and the burned token amount, including `pBurnAmount`, `itpBurnAmount`, and `ifpBurnAmount` as shown in line 297.

PoolLending.sol

```
265 function _withdraw(
266     address receiver,
267     uint256 nftId,
268     uint256 withdrawAmount
269 ) internal returns (WithdrawResult memory) {
270     PoolTokens storage tokenHolder = tokenHolders[nftId];
271 
272     uint256 itpPrice = _getInterestTokenPrice();
273     uint256 ifpPrice = _getInterestForwPrice();
274 
275     WithdrawResult memory interestResult;
276     if (withdrawAmount >= tokenHolder.pToken) {
277         interestResult = _claimAllInterest(receiver, nftId);
278         withdrawAmount = tokenHolder.pToken;
279     }
280 
281     require(withdrawAmount <= _currentSupply(),
"PoolLending/pool-supply-insufficient");
282 
283     uint256 itpBurnAmount = _burnItpToken(
284         receiver,
285         nftId,
```

```
286         (withdrawAmount * WEI_UNIT) / itpPrice,
287         itpPrice
288     );
289     uint256 ifpBurnAmount = _burnIfpToken(
290         receiver,
291         nftId,
292         (withdrawAmount * WEI_UNIT) / ifpPrice,
293         ifpPrice
294     );
295     uint256 pBurnAmount = _burnPToken(receiver, nftId, withdrawAmount);
296 
297     emit Withdraw(receiver, nftId, withdrawAmount, pBurnAmount, itpBurnAmount, ifpBurnAmount);
298     return
299         WithdrawResult({
300             principle: withdrawAmount,
301             tokenInterest: interestResult.tokenInterest,
302             forwInterest: interestResult.forwInterest,
303             pTokenBurn: pBurnAmount,
304             itpTokenBurn: itpBurnAmount + interestResult.itpTokenBurn,
305             ifpTokenBurn: ifpBurnAmount + interestResult.ifpTokenBurn,
306             tokenInterestBonus: interestResult.tokenInterestBonus,
307             forwInterestBonus: interestResult.forwInterestBonus
308         });
309 }
```

When the user withdraws all lending tokens (closing the lend position), the contract will claim the pending interest and burn `itpToken` and `ifpToken` in the `_claimAllInterest()` function called in line 277. During `_claimAllInterest()` execution, the `itpToken` and `ifpToken` will be burned if there is a profit from the interest.

However, in the `Withdraw` event, the burn amount of `itpBurnAmount` and `ifpBurnAmount` comes from the `_burnItpToken()` and `_burnIfpToken()` function, ignoring the burn result from the `_claimAllInterest()` function. This results in an invalid logging of `itpBurnAmount` and `ifpBurnAmount` values.

5.19.2. Remediation

Inspex suggests applying the result from `interestResult` with the `itpBurnAmount` and `ifpBurnAmount` amount to ensure the correct event logging, for example:

PoolLending.sol

```
265 function _withdraw(
266     address receiver,
267     uint256 nftId,
268     uint256 withdrawAmount
```

```
) internal returns (WithdrawResult memory) {
    PoolTokens storage tokenHolder = tokenHolders[nftId];

    uint256 itpPrice = _getInterestTokenPrice();
    uint256 ifpPrice = _getInterestForwPrice();

    WithdrawResult memory interestResult;
    if (withdrawAmount >= tokenHolder.pToken) {
        interestResult = _claimAllInterest(receiver, nftId);
        withdrawAmount = tokenHolder.pToken;
    }

    require(withdrawAmount <= _currentSupply(),
"PoolLending/pool-supply-insufficient");

    uint256 itpBurnAmount = _burnItpToken(
        receiver,
        nftId,
        (withdrawAmount * WEI_UNIT) / itpPrice,
        itpPrice
    );
    uint256 ifpBurnAmount = _burnIfpToken(
        receiver,
        nftId,
        (withdrawAmount * WEI_UNIT) / ifpPrice,
        ifpPrice
    );
    uint256 pBurnAmount = _burnPToken(receiver, nftId, withdrawAmount);

    emit Withdraw(receiver, nftId, withdrawAmount, pBurnAmount, itpBurnAmount +
interestResult.itpTokenBurn, ifpBurnAmount + interestResult.ifpTokenBurn);
    return
        WithdrawResult({
            principle: withdrawAmount,
            tokenInterest: interestResult.tokenInterest,
            forwInterest: interestResult.forwInterest,
            pTokenBurn: pBurnAmount,
            itpTokenBurn: itpBurnAmount + interestResult.itpTokenBurn,
            ifpTokenBurn: ifpBurnAmount + interestResult.ifpTokenBurn,
            tokenInterestBonus: interestResult.tokenInterestBonus,
            forwInterestBonus: interestResult.forwInterestBonus
        });
}
```

5.20. Unnecessary Functions in InterestVault

ID	IDX-020
Target	InterestVault
Category	Advanced Smart Contract Vulnerability
CWE	CWE-1164: Irrelevant Code
Risk	**Severity:** Info **Impact: None** **Likelihood: None**
Status	No Security Impact The FWX team has acknowledged this issue since it has no direct impact on the platform.

5.20.1. Description

The `pause()` and `unPause()` functions are used for pausing and unpausing functions in the `InterestVault` contract as shown below in the following source code:

InterestVault.sol

```
function pause(bytes4 _func) external onlyOwner {
    require(_func != bytes4(0), "InterestVault/msg.sig-func-is-zero");
    _pause(_func);
}

function unPause(bytes4 _func) external onlyOwner {
    require(_func != bytes4(0), "InterestVault/msg.sig-func-is-zero");
    _unpause(_func);
}
```

In the current implementation, the `InterestVault` contract is not using the `whenFuncNotPaused()` and `whenFuncPaused()` modifier in any function.

5.20.2. Remediation

Inspex suggests removing the `pause()` and `unPause()` functions from the `InterestVault` contract in case that they are not used.

5.21. Incorrect Logging for setForwAddress() Function

ID	IDX-021
Target	InterestVault
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Info **Impact: None** **Likelihood: None**
Status	Resolved The FWX team has resolved this issue as suggested by modifying the parameter that is passed to the `SetForwAddress` event.

5.21.1. Description

In the `setForwAddress()` function, the function emits the `SetForwAddress` event to indicate the new `forwAddress` as shown below in line 65:

InterestVault.sol

```
61 function setForwAddress(address _address) external onlyManager {
62     address oldAddress = forwAddress;
63     forwAddress = _address;
64 
65     emit SetForwAddress(msg.sender, oldAddress, tokenAddress);
66 }
```

However, parameters that are passed to the `SetForwAddress` event are not the new `forwAddress` state. This results in invalid logging of event, which the event is defined as shown below:

InterestVaultEvent.sol

```
event SetForwAddress(address indexed sender, address oldValue, address
newValue);
```

5.21.2. Remediation

Inspex suggests modifying the parameter that is passed to the `SetForwAddress` event, for example:

InterestVault.sol

```
function setForwAddress(address _address) external onlyManager {
    address oldAddress = forwAddress;
    forwAddress = _address;

    emit SetForwAddress(msg.sender, oldAddress, _address);
}
```

5.22. Improper Function Visibility

ID	IDX-022
Target	Membership
Category	Smart Contract Best Practice
CWE	CWE-710: Improper Adherence to Coding Standards
Risk	**Severity:** Info **Impact: None** **Likelihood: None**
Status	Resolved The FWX team has resolved this issue as suggested by changing the functions' visibility to `external` if they are not called from any internal function.

5.22.1. Description

Public functions that are never called internally by the contract itself should have external visibility. This improves the readability of the contract, allowing clear distinction between functions that are externally used and functions that are also called internally.

The following source code shows that the `pause()` function visibility in the `Membership` contact is set to public and it is never called from any internal function.

Membership.sol

```
146 function pause() public onlyOwner {
147     _pause();
148 }
```

The following table contains all functions that have public visibility and are never called from any internal function.

File	Contract	Function
Membership.sol (L:146)	Membership	pause()
Membership.sol (L:150)	Membership	unpause()

5.22.2. Remediation

Inspex suggests changing all functions' visibility to external if they are not called from any internal function as shown in the following example:

Membership.sol

```
function pause() external onlyOwner {
    _pause();
}
```

6. Appendix

6.1. About Inspex

CYBERSECURITY PROFESSIONAL SERVICE

Inspex is formed by a team of cybersecurity experts highly experienced in various fields of cybersecurity. We provide blockchain and smart contract professional services at the highest quality to enhance the security of our clients and the overall blockchain ecosystem.

Follow Us On:

Website	https://inspex.co
Twitter	@InspexCo
Facebook	https://www.facebook.com/InspexCo
Telegram	@inspex_announcement

CYBERSECURITY PROFESSIONAL SERVICE